Regression detection in software controlling industrial systems

Sébastien Salva
University Clermont Auvergne
LIMOS laboratory
Campus des Cézeaux, Aubière,
FRANCE
sebastien.salva@uca.fr

William Durand
University Clermont Auvergne
LIMOS laboratory
Campus des Cézeaux, Aubière,
FRANCE
william.durand@fr.michelin.com

Abstract: In the industry, testing production systems, i.e., systems transforming raw materials into products, is usually performed manually and is a long and error-prone task. The automation of the testing process could be initiated with the use of models. But, for this kind of system that has a long life span, models, when they exist, are seldom up-to-date. In this paper, we do not presume the availability of any model and we propose a method to automatically test the CIM2 level of production systems i.e., the software controlling them, by combining two approaches: model inference and passive testing. Using a set of events collected from a production system, our approach combines the notions of expert system, formal models and machine learning to infer symbolic models while preventing over-generalisation (i.e., the models should not capture more behaviours than those possible in the real system). These models are then used as specifications to passively test the conformance of other production systems. We define conformance with two complementary implementation relations. The first relation is based upon the classical trace-preorder relation. The second one is a weaker relation, less strict on the parameter values found in the traces of the system under test. With the collaboration of the manufacturer Michelin, we evaluated our approach on a real production system and show that it can be used in practice to quickly generate models and to test new production systems.

Key–Words: Software engineering, Maintenance, Model generation , Passive testing , Industrial systems

1 Introduction

In this paper, we study formal testing applied to industrial systems such as those of our partner Michelin, one of the three largest tire manufacturers in the world. We focus on production systems, made up of heterogeneous devices (machines, tools, sensors, robots etc.), interconnected with robust networks and controlled by a software in a factory. To avoid damaging the devices of a production system, the software controlling the devices is often tested manually with simulations, which replicate machine behaviours or human operations. This testing phase usually requires a long period of time, from some weeks up to some months.

A lot of works already deal with the test of (distributed) event-based systems and propose to automate partially or completely the testing stage. Therefore, why is it difficult to abandon manual testing with this kind of system? Active testing is the conventional way to test the conformance of *black-box* systems [25, 8]. This approach requires a tester that stimulates the system under test with test cases to observe its reactions. To automate the testing process (especially when the system includes many different scenarios), test cases are generated from a (formal) model to be later executed by the tester. Active testing faces limitations though, particularly in the manufacturing context:

- to stimulate the system, it must be shut-down, interrupted or reset for some time. Resetting such a system is difficult and often long. Furthermore, frequent interruptions may lead the devices to abnormal functioning up to make them break down,
- another important drawback of active testing lies in the basic need of a specification. Writing a specification is known as a long and error-prone task. Furthermore, production systems belong to the specific category of systems that have a life span of many years, (sometimes up to 20 years). These systems are incrementally updated (physical devices, applications), but usually not the models. The latter become out-dated in the long run and can no more be used for testing. This is a common problem with documentation in general, and it often implies rather under-specified or not documented systems that become awkward to maintain because of lack of understanding.

This paper tackles the problem of testing such systems, without disturbing them, and without hav-

ing any specification. We focus on the second level of the CIM approach (Computer Integrated Manufacturing [40]), which corresponds to the software controlling the physical devices of a production system. We firstly propose a pragmatic and theoretically well-founded method for inferring models from production systems. These models are then considered as reference specifications and can be used to test the same updated system or new ones.

Specifically, given a running production system, we start by generating models by means of the observations collected among devices and the applications controlling them. We call these observations *production events*. These models describe the functional behaviours of the system, and may serve different purposes, e.g., testing a production system. Several model inference methods can be found in the literature, but most build over-approximated models, i.e., models capturing more behaviours than those possible in the real system. In our context, we want to prevent the generation of over-approximated models as their use for testing often increases the risk to make emerge false positive results (wrong detections of errors). As a consequence, we propose our own model inference method, based upon an expert system to materialise the knowledge of experts of the system and model transformations. From a production event set, we generate formal models called Symbolic Transition systems (STSs, [19]), by means of inference rules, which can be adapted to match different kinds of systems. In addition, as production systems can produce millions of events on a daily basis, we made a scalable model inference engine, which builds models in reasonable time.

Afterwards, we use these models to passively test the conformance of another production system. Generally speaking, passive testing represents an alternative approach to check the reliability (in terms of conformance, robustness, security, etc.) of implementations, without altering their normal functioning, i.e., by passively observing their behaviours. It has been applied for protocol testing [24, 5, 12], runtime verification [18], etc. In our context, we devised a passive tester, which checks whether the observed executions of a (second) production system under test adhere to the behaviours of a model inferred from a first system. The passive tester collects the execution traces (sequences of events) of the system under test by reusing the building blocks of the model generation engine. It produces a set of traces having the same level of abstraction as those considered for inferring the model and uses them to check whether the system under test conforms to the model. Conformance is defined without ambiguity with two implementation relations. Although several classical implementation relations could be considered, our relations are written to take into account the use of under-approximated (partial) models. The passive tester algorithm implements these relations.

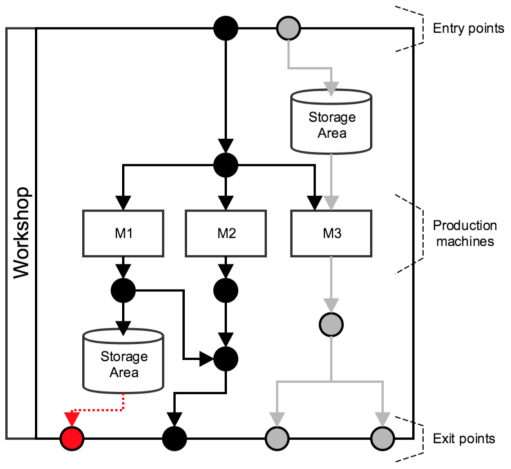

Figure 1: Simplified representation of a workshop

Our approach has been implemented in a framework called *Autofunk* (for Automatic Functional testing tool), which combines both model generation and testing engines. In the remainder of this paper, we present the theoretical background that we considered to devise Autofunk and preliminary results on real production systems of Michelin.

Paper organisation:
The paper is structured as follows: Section 2 introduces the background of the production systems of Michelin. We propose an overview of the different flavours of model inference and passive testing techniques. We also discuss our motivations. Section 3 presents an overview of our approach. We describe the practical assumptions that guided the design of Autofunk. Then, the three next sections present the the theoretical aspects of Autofunk: we recall some basic concepts and definitions about the STS formalism in Section 4, Section 5 describes the model inference stage and Section 6 the passive testing stage. We introduce an evaluation of some experiments made on a real production system, in Section 7. We conclude the paper and outline some perspectives in Section 8.

2 Background

2.1 Michelin production systems

Michelin is a worldwide tire manufacturer and designs most of its factories, production systems, and related software. Like many other industrial companies, Michelin follows the Computer Integrated Manufac-

turing (CIM) approach, using computers and software to control the entire manufacturing process and acquire data. The CIM approach segments the manufacturing process and production strategies into several hierarchical levels: CMI1 is the device level, CMI2 includes all the applications that monitor and control devices. Levels 3 and 4 focus on the factory management.

A factory includes workshops, each devoted to a step of the tire building process, e.g., tire assembling (assembling the components onto a tire drum) or curing (applying pressure on assembled tires in molds to give their final shapes). A workshop gathers devices, branch points, conveyor belts and human operators that perform specific actions (removal of products to assess their qualities, etc.). A workshop is controlled by a set of CIM2 applications (except for the operators): every order (move, stop, change state, etc.), product modifications or alerts passing among industrial devices and software are materialised with messages that we call *production events*. These applications are continuously updated and sometimes replaced, for instance when the physical configuration of the workshop is modified, when new machines are added, when bugs are detected, or when it is decided that applications are too old and are no more maintainable.

As depicted in Figure 1, at the workshop level, we observe a continuous stream of products following assembly lines from specific *entry points* to *exit points*, i.e., where products go to reach the next step of the manufacturing process. Some factories produce over 30,000 tires a day, resulting in thousands of production events at the CIM2 level, which are collected and persisted in databases.

Production systems are tested when they are set up and every time they are updated with new applications and parameters, new devices, etc. We do not focus on the device level here (CIM1), but on the CIM2 level (although physical devices are tested too). For readability, when we refer to production systems in the remainder of the paper, we actually focus on the software of the CIM2 level, which acquires and sends production events to the devices.

For testing a production system, Michelin engineers firstly build simulations by mocking most of the devices. Then, they run hundreds of scenarios composed of production events, collect all the observable production events and manually inspect them to detect abnormal behaviours. As simulations are not sufficient to run all the possible scenarios, production events are again collected when the system is running, and events are scrutinised every time an issue is detected. This manual testing process can be followed for a long period time, depending on the modifications made on the system (up to 6 months). Michelin wished to partially automate this phase to:

- quicker detect potential regressions when CIM2 applications are modified or when devices are replaced to ensure that they are interoperable with the current application versions,
- test an updated system, different from the original one (hence composed of new applications) to get its set of traces expressing new behaviours. Michelin engineers could later focus on them and seek for potential faults,
- reduce the testing delay,
- get exact models in the sense that they do not capture more behaviours than those possible in the real system. These models could be studied to diagnose issues of the production system.

The main problem faced by Michelin lies in the lack of up-to-date documentation. Indeed, the average lifetime of the applications deployed in their factories is 20 years. During this long lifetime, applications are updated many times independently in every factory all over the world, potentially highlighting different behaviours and features. Initially, these applications are documented with models, which become outdated in the long run. Furthermore, even if a lot of effort is put into standardising applications and development processes, different programming languages and frameworks are still used by development teams, making difficult to focus on a single technology. This application set appears too disparate and insufficiently documented to apply conventional testing techniques. This is why our industrial partner firstly needs a safe way to infer up to date models, regardless of the underlying technical details, and without having to rely on any existing documentation.

In addition, Michelin engineers need of a scalable tool since a production system produces thousands of tires a day, along with thousands of production events. When an issue is detected in a production system or when the latter is getting jammed, they are interested in getting models as quickly as possible to help them diagnose failure causes.

2.2 Related work and motivation

Our approach is mainly founded upon two research fields, model inference and passive testing. Several papers dealing with these domains were issued in the last decade. We present some of them related to our work, and introduce some key observations.

Model inference

Model inference originates from approaches dedicated to language learning started in the 1970's with Gold [20]. Model inference can be defined as *a set of methods that infer a specification by gathering and analysing system executions and concisely summarising the frequent interaction patterns as state machines that capture the system behaviour* ([1]). These models, even if partial, can be examined by a engineer, to refine the specification, to identify errors, or can be considered for different kinds of analyses, etc. The model generation can be performed from different kinds of data samples such as affirmative/negative answers ([3]), execution traces ([22]), documentation ([43]), source code ([32, 31]) or network traces ([4]). After reviewing the literature, we observed that two main categories of methods emerge, which we call active and passive methods. The first category contains methods that repeatedly query systems or humans to collect positive or negative observations. These are analysed to build models and eventually to perform other queries. Many active inference techniques have been conceived upon two concepts, the $\mathcal{L}^*$ algorithm of [3] and incremental learning (e.g., [15]).

In the context of production systems, this active functioning requires a lot of effort to be practical though. The sending of queries or the use of active testing techniques on production systems usually requires to frequently reset it, which is a long and costly process. This is why we prefer focusing on the passive inference category. It includes techniques that infer models from a given set of samples, e.g., a set of execution traces. Since there is no interaction with the system to model, these techniques are said passive or offline. Models are often constructed by representing sample sets with automata whose equivalent states are later merged. A substantial part of the papers covering this topic propose approaches either based upon event sequence abstraction or state-based abstraction to infer models.

With event sequence abstractions, the detail level of the models is raised by merging the states having the same event sequences. This process stands on two main algorithms: *kTail* [7] and *kBehavior* [29]. kTail generates models from trace sets with two steps. First, it builds a Prefix Tree Acceptor (PTA), which is a tree whose edges are labelled with the events found in traces. Then, kTail transforms the PTA into a Finite State Automaton (FSA) by merging every pair of states if they exhibit the same future of length k, i.e., if they have the same set of event sequences having the maximum length k, accepted by the two states. kBehavior generates FSA from a set of traces by taking one trace after one and by completing the FSA such that it accepts the trace. More precisely, whenever a new trace is submitted to kBehavior, it identifies the sub-traces that are accepted by sub-automata in the current FSA (the sub-traces must have a minimal length k, otherwise they are considered too short to be relevant). Then, kBehavior extends the model with the addition of new transitions that suitably connect the identified sub-automata, producing a new version of the model, which now accepts the entire trace. Both Algorithms were enhanced to support events combined with data values [28].

The approaches, which use state-based abstraction, adopted the generation of state invariants to define equivalence classes of states that are combined together to form final models. The *Daikon* tool [17] were originally proposed to infer invariants composed of data values and variables found in execution traces. An invariant is a property that holds at a certain point or points in a software. An invariant generator mines the data found in traces, and then reports properties that are always *true*. Several works were proposed to infer models from traces produced by software components ([38, 22]) or source codes ([42]).

Passive testing

Observing the behaviour of an implementation and testing if it adheres to a given user-provided specification has been referred under different names such as passive testing or runtime verification. With runtime verification, specifications are usually written with CTL or LTL properties, which is out of the scope of the paper. We prefer referring to [26] for introducing runtime verification. Passive testing (and runtime verification) offers the advantage to not disturb the implementation under test by collecting observations or samples and by checking if these meet a specification or properties. Several works, dealing with the passive testing of protocols or components, have also been proposed over the last decade. We propose to group some of them related to our work in two different categories:

- *Invariant satisfiability:* invariants represent properties that are always true. An invariant is constructed by hand, and later checked on a set of traces collected from an implementation. This approach is very similar to runtime verification and allows the test of complex and formal properties. It gave birth to several works, e.g., [11, 10, 6, 30]. For instance, the passive testing method presented in [11] aims to test the satisfiability of invariants on Mobile ad-hoc network (MANET) routing protocols. Different steps are required: definition of invariants, extraction of

execution traces with sniffers, verification of the invariants on the trace set. Other works focused on Component-based System Testing: in this case, passive methods are usually used for conformance or security testing. Andrès et al. presented another methodology to perform the passive testing of timed systems [2]. The paper gives two algorithms, which check whether timed invariants hold on logs recorded from an implementation under test;

- *Forward checking:* implementation reactions are observed by a tester, which detects incorrect behaviours by covering a model every time a new event is collected [24, 37]. For instance, Lee et al. proposed some methods dedicated to wired protocols in [24]. Protocols are modelled with Event-driven Extended Finite State Machines (EEFSM), which are specialised FSM composed of variables and constraints over these variables. Several algorithms on the EEFSM model are provided as well as their applications on the protocols TCP and Open Shortest Path First (OSPF).

2.2.1 Key observations and motivations

After having studied both research fields and discussing with our industrial partner, it quickly turned out that passive inference appeared to be a good fit. Nevertheless, the proposed approaches still reflect some limitations that plague the final quality of the models. Most of the model inference techniques are over-approximating system behaviours, i.e., models often admit more behaviours than those observed. When models are employed for verification or testing, over-approximated models often bring false positives. Indeed, infeasible test cases can be generated from these models, i.e., test cases that cannot be executed or that expect observations the system cannot produce. Such test cases give incorrect verdicts. We observed that over-approximation often comes from the state merging process, which combines the states having the same properties. Specifically, this issue comes from the state equivalence relations (k-future, congruence equivalences, etc.) that raise the abstraction level of the model. Furthermore, most of the above algorithms have a complexity polynomial in time with respect to the model size or require a polynomial number of queries. However, we observed that only few methods [42, 31] focus on scalability and propose algorithms that can take huge event sets as inputs and still infer models quickly. To do so, these use a context-specific state merging process.

Based on these observations, we chose to devise a (context-specific) model inference approach that aims at recovering models as Symbolic Transition Systems (STS [19]) from production event sets. As models are used for testing, we want to prevent (control) their over-generalisations. This approach is hence initially based upon trace abstraction and model compression to avoid the construction of models composed of over-approximations. Then, we remove the information related to products, which we call normalisation. The originality of the model construction resides in the combination of an expert system to encode expert knowledge (given by Michelin engineers or found in documentation) and of transition systems to embrace formal tools. This means that the STS transformations and compression, called STS reduction, are defined with inference rules thanks to the STS theory, and are triggered by the same expert system. The STS reduction, is based on an event sequence abstraction. We also show that our approach is scalable: it can take millions of production events and still build models quickly thanks to our specific state merging process.

Concerning passive testing, we noticed that the above techniques assume having either complete specifications encoding all the correct behaviours, or invariants. The implementation relations are not specifically tailored to support partial inferred models, which are neither complete specifications nor properties. The passive testing technique that we propose is founded on two complementary implementation relations to define conformance while taking into consideration the use of under-approximated (partial) models. The first relation is based upon the classical preorder relation but only applied on the ”filtered” traces of the system under test. The second one is a weaker relation less strict on the parameter values found in the traces. The tester algorithm is based upon these relations. Additionally, it reuses several blocks of the model generation step to build the traces of the system under test.

Finally, we introduced in [16] a premise of this passive testing method. This paper extends it by revisiting the theoretical aspects of model inference and passive testing. We give new definitions and propositions on the two implementation relations to give one tester algorithm using both and to prepare the proof of its soundness. We also evaluate the implementation of our approach on a real manufacturing system.

3 Approach overview

After several discussions with people at the Michelin company and after having studied the functioning of a factory, production system log files and internal documentation, we concluded that the applications of the CIM2 level, used to operate a production system,

continuously produce raw data adequately rich to deduce the system behaviours. Indeed, the messages found in log files carry a lot of interesting information (labels, parameters) that can be interpreted to understand how a whole industrial system behaves. Michelin engineers are actually able to read them to infer high level actions. These messages are not tied to any programming language or framework. Furthermore, these are exchanged over a network layer that guarantees (synchronous) ordering and delivery. Studying the behaviours of production systems at this level makes some assumptions emerge that have been considered in the design of our framework:

- Black-box systems: production systems are seen as black-boxes from which a large set of production events can be passively collected at the CIM2 level. Such systems are compound of assembly lines fragmented into several devices and sensors, synchronised by a global clock. A production system has one or more entry and one or more exit points;
- Production events: a production event of the form $a(\alpha)$ includes a distinctive label a along with a parameter assignment α. Two production events $a(\alpha_1)$ and $a(\alpha_2)$ having the same label a must own assignments over the same parameter set. Network protocols guarantee synchronous communications and the event ordering with timestamps assigned to a parameter denoted $time$, which takes values from a global clock. A specific parameter, denoted $point$, stores the physical location of devices. For instance, the parameter $point$ can be assigned to coordinates or to device identifiers;
- Traces identification: execution traces are sequences of production events $a_1(\alpha_1)...\ a_k(\alpha_k)$. A trace is identified by a specific parameter that is included in all the event assignments of a trace. In this paper, this identifier is denoted with pid and identifies products, e.g., tires at Michelin. At the same time, we cannot have two different traces (for two products) having the same pid;
- Event delivery: network protocols guarantee the event delivery and an assembly line is conceived in such a way that it does not have deadlock states except when a product exits the line.

We present in this section an overview of our approach called Autofunk. The next sections will detail every step briefly mentioned here. Autofunk, takes as inputs the production events of a first production system to infer its model and then passively tests a second production system to ensure that it is conforming

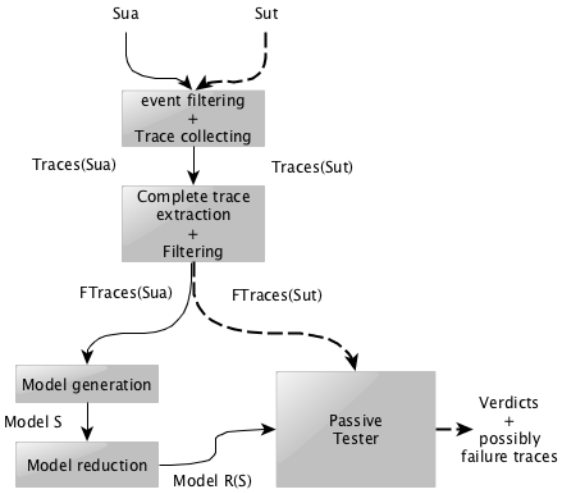

Figure 2: Autofunk overall architecture

to the model. These two parts are depicted in Figure 2 (model inference with solid lines, passive testing with dashed lines).

In this paper, we focus on models called Symbolic Transition Systems (STS). The STS formalism is widely used in the verification and testing areas to describe large and complex event-driven systems. Other models, e.g., the EFSM model [13] bring interesting features but STSs offer another advantage with this work. We can indeed benefit from the well established testing theory given in [36, 19] and hence define inference rules to express STS operations and transformations. This last aspect helps combine the two different areas we consider in this paper to infer models: formal models and expert systems. Indeed, human expert knowledge is materialised with inference rules applied on knowledge bases (production events, traces, STSs) e.g., to filter out production events. In the same way, STS transformations are given with inference rules applied on the same knowledge bases. The same inference engine is called whatever the purpose of the rules.

Autofunk recovers models (solid lines in Figure 2), which represent the functional behaviours of a system under analysis denoted Sua. This step is mainly framed upon an inference engine to build formal models from production events by means of successive transformations. Initially, Autofunk takes log files, composed of messages, collected from Sua. The CIM2 level and the network protocols used in Michelin's factories guarantee that the messages can be collected by a log server while preserving the message ordering. These messages are formatted as a set of production events of the form $a(\alpha)$ with a a label, and α a parameter assignment. Production events are filtered

to produce a first set of traces denoted $Traces(Sua)$. Products could stay in a workshop for days and even weeks, i.e., longer than the time period considered for collecting production events. From our point of view, this means that we can likely collect *incomplete* traces in $Traces(Sua)$, that is the traces that are not obtained from an entry point of the production system or that not end in one of its exit points. Such traces express incorrect behaviours in the sense that they do not reflect the complete functioning of assembly lines. Furthermore, we observed that the CIM2 applications may send the same request multiple times to physical devices until they get an acknowledgement. Repetitions of sequence of events, that we call *repetitive patterns* of events, may happen a multitude of times, depending on the use of the load of the devices. Most of these repetitive patterns of events are considered as noise by the Michelin engineers because these events do not represent new behaviours of an assembly line. This is why Autofunk detects repetitive patterns, provides them to an engineer so that it may chose to keep some of them as desired. The traces including the other repetitive patterns (those not chosen) are filtered out. The same process shall be applied for testing. $Traces(Sua)$ is therefore filtered to only keep the *filtered traces* of $Traces(Sua)$, denoted $FTraces(Sua)$, namely the traces related to products not composed of repetitive patterns, which are obtained from an entry point of the production system to one of its exit points.

The number of entry or exit points depends on the configuration of the assembly lines and may varies among the factories. Entry and exit points can be given by an expert of the system (especially when the collected set of production events is too small). But, as the set of production events is usually large, we have chosen to apply a machine learning technique on the production events to automatically deduce the entry and exit points. The technique employed here is K-means [21] on the occurrences of every potential entry point (found in every first event of the trace of $Traces(Sua)$) and exit points (found in every last event of the traces). Section 5.2 gives more details about this step, which helps automate the model inference process at the most, without being time consuming.

The set $FTraces(Sua)$ is then partitioned into subsets $(ST_1, ..., ST_n)$, one set for each entry point of the system, to later build in parallel one model per entry point. We construct the model S, which is composed of the list of STSs $(\mathcal{S}_1, ..., \mathcal{S}_n)$ (with n the number of detected entry points). Here, we also remove all the parameter assignments related to products (product id, timestamps) to express more general behaviours (not related to products). We call this step STS normalisation. Usually, it turns out that S has a (very) large set of transitions, which could negatively influence the complexity of the testing stage. That is why our framework adds a reduction step, aiming at diminishing the first model S into a second one, denoted $R(S)$.

The second part of *Autofunk* relates to the testing phase (dashed lines in Figure 2). It takes a model $R(S)$ inferred from the reference system Sua and production events collected from another system under test Sut. Sut can refer to the same production system as Sua, which has been updated. In this case, testing comes down to check that changes have not introduced new faults (regression testing). Sut can also be a new system in a new factory, which should behave as Sua. We have chosen to formally define conformance with two implementation relations between Sut and $R(S)$. The first one, denoted $\leq_{ft}$ checks whether the filtered traces of Sut, in $FTraces(Sut)$, belong to $Traces_{Pass}(R(S))$, which is the set of traces of $R(S)$ expressing correct behaviours. As $R(S)$ is a partial model that does not necessarily encode all the possible behaviours of Sua, this relation may reject some potential correct implementations. This is why we define a complementary implementation relation, denoted $\leq_{mft}$, which means that Sut is correct iff for each execution trace t, all its parameter assignments can be found into several traces of $Traces_{Pass}(R(S))$ having the same sequence of events as t (instead of only one trace with the first relation). The passive tester reuses some modules of the model inference step (event formatting, trace filtering) to build $FTraces(Sut)$ and implements these two relations. It produces the verdicts "Pass$\leq_{ft}$", "Pass$\leq_{mft}$" if the relations hold. Additionally, when a faulty implementation is detected, it returns its possibly failure traces i.e., the traces of Sut that cannot be extracted from the model $R(S)$. Here, we talk about possibly failure traces because our models are partial and do not necessarily encode all the possible correct behaviours. Hence, these traces should be later inspected to ensure whether they capture wrong behaviours.

Before covering the formal aspects of the model generation and the testing of production systems, we recall some notations and definitions about the STS model.

4 Model Definition and Notations

In this paper, we focus on models called Symbolic Transition Systems (STS) ([19]) to represent how production systems behave.

4.1 Symbolic Transition Systems

A STS is a kind of symbolic automaton compound of states called *locations*. Transitions between locations are labelled with events including a label and parameters. One can also find guards and variable assignments.

Definition 1 (Variable assignment) *We assume that there exist a finite domain of values denoted D and a variable set X taking values in D. The assignment of variables in $Y \subseteq X$ to elements of D is denoted with a mapping $\alpha : Y \rightarrow D$.*
We denote D_Y the assignment set over Y. For $y \in Y$, $\alpha(y)$ returns the assignment of the y variable. We also denote id_Y the identity assignment over Y.

For instance, $\alpha = \{x := 1, y := 3\}$ is a variable assignment of $D^{\{x,y\}}$. $\alpha(x) = \{x := 1\}$ is the variable assignment related to the variable x.

Definition 2 (STS) *A Symbolic Transition System (STS) is a tuple $(L, l0, V, V0, I, \Lambda, \rightarrow)$, where:*

- *L is the finite location set, with l_0 being the initial one,*
- *V is the finite set of internal variables, I is the finite set of parameters,*
- *V_0 is the initial condition, a predicate with variables in V,*
- *Λ is the finite set of symbolic events $a(p)$, with $p = \langle p_1, ..., p_k \rangle$ a finite tuple of parameters in $I^k (k \in \mathbb{N})$,*
- *$\rightarrow$ is the finite transition set. A transition $(l_i, l_j, a(p), G, A)$, from the location $l_i \in L$ to $l_j \in L$, also denoted $l_i \xrightarrow{a(p),G,A} l_j$, is labelled by:*
  - *an event $a(p) \in \Lambda$, with $p = \langle p_1, ..., p_k \rangle$,*
  - *a guard G, which is a predicate with variables in $V \cup \{p_1, ..., p_k\}$ that restricts the firing of the transition. For simplicity (and since this is sufficient in our context), we restrict to the guards of the form:*
    $G \rightarrow PG \mid G \; op \; G$,
    $PG \rightarrow Variable == Constant$,
    $op \rightarrow \wedge \mid \vee$,
  - *internal variables are updated with the assignment function A of the form $(x := A_x)_{x \in V}$, A_x is an expression over $V \cup \{p_1, ..., p_k\}$.*

Below, we define some notations on STSs. In particular, we use the notion of projection on guards, denoted $Proj_X(G)$, which aims to only keep the equalities of G using the variables of the set X. A projection comes down to eliminate the equalities using the variables in $(I \cup V) \backslash X$. For the definition of variable elimination (a.k.a. forgetting), we refer to [23, 39]. For instance, in propositional logic, removing a literal l from a formula F, is given by $ForgetLit(F, \{l\}) = F_{l \leftarrow 1} \vee (\neg l \wedge F)$ [23].

Definition 3 *Given a STS $\mathcal{S} = (L, l0, V, V0, I, \Lambda, \rightarrow)$ and l, $l' \in L$, we use the following notations:*

- $l_1 \xrightarrow{(a_1,G_1,A_1)...(a_n,G_n,A_n)} l_{n+1} =_{def} \exists l_i, l_{i+1}, a_i, G_i, A_i \; (1 \leq i \leq n) : l_1 \xrightarrow{a_1,G_1,A_1} l_2, ..., l_n \xrightarrow{a_n,G_n,A_n} l_{n+1}$;
- *l is a deadlock location iff $\neg \exists l', a(p), G, A : l \xrightarrow{a(p),G,A} l'$;*
- $l \xrightarrow{a(p),G} l' =_{def} \exists, a(p), G, A = id_V : l \xrightarrow{a(p),G,id_V} l'$;
- *$Proj_X(G)$, the projection of the guard G over the variable set $X \subseteq I \cup V$, which eliminates from G the equalities on the variables of $(I \cup V) \backslash X$.*

The use of symbolic variables helps describe infinite state machines in a finite manner. This potentially infinite behaviour is represented by the semantics of a STS, given in terms of Labelled Transition System (LTS). The LTS semantics can be assimilated to a valued automaton, which is often infinite: the LTS states are labelled by internal variable assignments, and transitions are labelled by valued events, composed of parameter assignments. The semantics of a STS $\mathcal{S} = (L, l0, V, V0, I, \Lambda, \rightarrow)$ is the LTS $||\mathcal{S}|| = (Q, q_0, \sum, \rightarrow)$ composed of valued states in $Q = L \times D_V$, $q_0 = (l_0, V_0)$ is the initial one, $\sum$ is the set of valued events, and $\rightarrow$ is the transition relation.

The complete definition of the relation between a STS and its LTS semantics is given in [19]. For simplicity, we only give its insight in this paper. For a STS transition $l \xrightarrow{a(p),G,A} l'$, we have LTS transitions of the form $(l, v) \xrightarrow{a(p),\alpha} (l', v')$ with v an assignment over the internal variable set if there exists a parameter value set α such that the guard G evaluates to true with $v \cup \alpha$. Once the transition is fired, the internal variables are assigned with v' derived from the assignment $A(v \cup \alpha)$.

From the LTS semantics, one can derive runs and traces, which reflect the concrete functioning of the system modelled with $\mathcal{S}$ and $||\mathcal{S}||$:

Definition 4 (Runs and traces) *Let $\mathcal{S}$ be a STS and $||\mathcal{S}|| = (Q, q_0, \sum, \rightarrow)$ be its LTS semantics.*

- *A run $q_0 a_0(\alpha_0)...q_{k-1}a_{k-1}(\alpha_{k-1})q_k$ is an alternate sequence of states and valued events such that: $\exists q_i, q_{i+1}, a_i, \alpha_i (0 \leq i \leq k-1) : q_0 \xrightarrow{a_0(\alpha_0)} q_1 ... \ q_{k-1} \xrightarrow{a_k(\alpha_k)} q_k \in \rightarrow^*$.*
  $Runs(\mathcal{S}) = Runs(||\mathcal{S}||)$ is the set of runs found in $||\mathcal{S}||$.
  $Runs_F(\mathcal{S})$ is the set of runs that end in a state q of $F \times D_V$ with $F \subseteq L$.
- *the trace of a run $r = q_0 a_0(\alpha_0)...q_{k-1}a_{k-1}(\alpha_{k-1})q_k$, denoted $Trace(r)$ is the sequence $a_0(\alpha_0)...a_{k-1}(\alpha_{k-1})$.*
  $Traces_F(\mathcal{S}) = Traces_F(||\mathcal{S}||) = \{Trace(r) \mid r \in Runs_F(\mathcal{S})\}$.

4.2 Production system models

Intuitively, our STS generation is performed by retro-engineering a production system from its traces by referring to the above definitions: traces are collected from production systems, then filtered out, and transformed into runs. From these, we construct STSs.

For simplification and to later help in the parallelisation of the model inference process, we have chosen to segment a production system model into several STSs, one STS per entry point. In production events, the parameter $point$ stores device locations and can be used to describe entry and exit points. Additionally, we observed that devices may query CIM2 applications multiple times until they get an acknowledgement. As stated earlier, we call these sequence of events, *repetitive patterns* of events. With all these characteristics, it is now possible to propose a definition of a production system model:

Definition 5 (Production system) *A production system S is a n-tuple ($n > 0$) $S = (\mathcal{S}_1, \ldots, \mathcal{S}_n)$ with $\mathcal{S}_i = (L_{\mathcal{S}_i}, l0_{\mathcal{S}_i}, V_{\mathcal{S}_i}, V0_{\mathcal{S}_i}, I_{\mathcal{S}_i}, \Lambda_{\mathcal{S}_i}, \rightarrow_{\mathcal{S}_i})$ $(1 \leq i \leq n)$ such that $\forall\ l0_{\mathcal{S}_i} \xrightarrow{a(p),G,A} l \in \rightarrow_{\mathcal{S}_i}$: $Proj_{\{point\}}(G) = (point == v)$, $(point := v) \in Entry(S)$.*

$Entry(S)$ (resp. $Exit(S)$) denotes the set of assignments on the variable $point$ modelling the entry points (resp. the exit points) of S. $Card(Entry(S)) = n$.

$Pattern(S)$ is the set of repetitive patterns of S, $Pattern(S) = \{a_1(\alpha_1) \ldots a_k(\alpha_k) \mid \forall (1 \leq j \leq k), a_j(p_j) \in \bigcup_{1 \leq i \leq n} \Lambda_{\mathcal{S}_i}$ and $\alpha_j \in D_{p_j}\}$.

We also denote $Traces(S) = \bigcup_{1 \leq i \leq n} Traces(\mathcal{S}_i)$.

We are now ready to expose how to infer production system models from collections of production events.

5 Model inference of production systems

The model inference part aims at recovering models representing the functional behaviours of a system under analysis. To reason about this system, we now assume that it can be modelled by a (unknown) LTS, denoted Sua. This assumption allows to later denote its trace set with $Traces(Sua)$ and so on.

The originality of the model inference part lies in the combination of the notion of expert system with the STS formalism. Our framework uses an expert system adopting a forward chaining. Such a system separates the knowledge base, a.k.a. facts, from the reasoning: the former is expressed with data and the latter is defined with inference rules that are applied on the facts. All information handled by Autofunk (events, traces, transitions, models) are then modelled with bases of facts. We rely upon two kinds of inference rules to infer STSs. On the one hand, we have rules capturing the knowledge of a human expert or found in documentation. These are expressed with rules of the form *When condition, Then action(s)* applied on facts. On the other hand, the remaining rules relate to STS transformations, which can be formally defined with inference rules as well. The possibility to change rules for matching other kinds of systems is a manifest benefit of using inference rules to infer models. Nevertheless, such rules have to be conceived with care. Indeed, they have to be triggered a finite number of times to ensure the model inference termination and must always give identical results with the same bases of facts. To reach that purpose, we assume that inference rules and knowledge bases meet these hypotheses:

Model inference assumptions:

1. inference rules are Modus Ponens (simple implications that lead to sound facts if the original facts are true);

2. the facts in knowledge bases have an Horn form (facts with at most one positive literal, e.g., simple facts).

These assumptions guarantee that the resolution of the inference rules (based on Modus Ponens) with

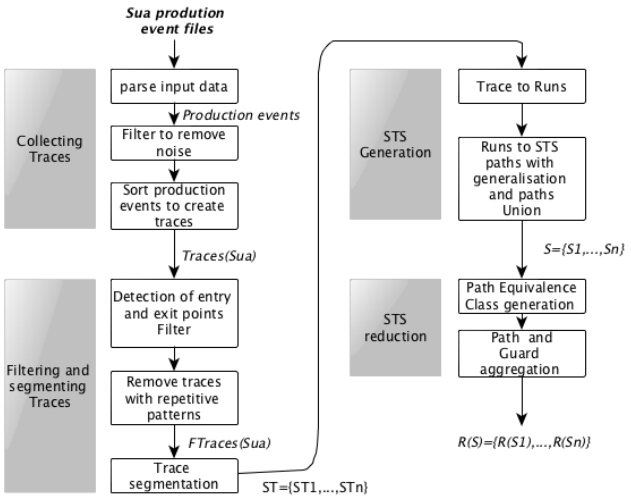

Figure 3: Model inference detailed steps

```
17-Jun-2016 23:29:59.00|INFO|New File
17-Jun-2016 23:29:59.50|17011|MSG_IN
[nsys: 1] [nsec: 8] [point: 1] [pid: 1]
17-Jun-2016 23:29:59.61|17021|MSG_OUT
[nsys: 1] [nsec: 8] [point: 3] [tpoint: 8] [pid: 1]
17-Jun-2016 23:29:59.70|17011|MSG_IN
[nsys: 1] [nsec: 8] [point: 2] [pid: 2]
17-Jun-2016 23:29:59.92|17021|MSG_OUT
[nsys: 1] [nsec: 8] [point: 4] [tpoint: 9] [pid: 2]
```

Figure 4: Production event examples

a knowledge base (in Horn form) is sound and complete. These assumptions will be used to discuss about the soundness and complexity of the model inference step in Section 5.5.

The model inference steps followed by Autofunk are depicted in Figure 3. Each grey box in the figure is detailed one after the other below.

5.1 Trace collecting

Autofunk starts by collecting production events from databases gathering the events passing through the network of the system under analysis. In this way, Sua is not disrupted since production events are collected by its logging system. A production event is mainly compound of a label along with kinds of variable assignments.

An example of log file is given in Figure 4. It includes simplified production events similar to those extracted from the Michelin logging system. *INFO*, *7011* and *17021* are labels that are accompanied with assignments of variables, e.g., $nsys$ that indicates an industrial device number or $point$ that gives the device position in a workshop. With real events, there

```
rule "Remove INFO events"
when:
$a: ValuedEvent(assignment.valueOf("type") ==
        TYPE_INFO)
then
retract($a)
end
```

Figure 5: Inference rules example for filtering

are around 20 parameters. Such a format is specific to Michelin but other kinds of events could be considered by updating the Autofunk parsing module.

Production events are stored in a knowledge base and formatted as valued events $a(\alpha)$, with a a label and α a parameter assignment. Thereafter, this base is filtered to remove the production events that are considered as unnecessary to describe the functional behaviours of Sua. These are filtered by means of inference rules of the form: *When* $a(\alpha)$*, condition on* $a(\alpha)$*, Then retract(*$a(\alpha)$*)*. Figure 5 shows a rule example applied on Michelin systems. This rule is written with the *Drools* [1] formalism. Drools is a rule-based expert system using knowledge bases expressed with Java objects. This rule removes the production events that hold the *INFO* label. Indeed, experts confirmed us that it does not worth keeping this kind of event since they do not express a system behaviour, but network status.

From this filtered production event base, we construct traces by linking together the production events $a(\alpha)$ holding the same trace identifier pid, and by ordering them with respect to their timestamps assignments. We call the resulting trace set $Traces(Sua)$:

Definition 6 (**$Traces(Sua)$**) *Given a system under analysis* Sua*,* $Traces(Sua)$ *denotes its trace set.*

$Traces(Sua)$ *includes finite traces i.e., finite sequences of production events of the form* $a_1(\alpha_1)....a_k(\alpha_k)$ *such that:* $\exists$ *unique* $v \in D, \forall \alpha_i (1 \leq i \leq k) : \alpha_i(pid) = (pid := v)$.

5.2 Trace segmentation and filtering

Products could stay in a workshop for days and even weeks. Sometime, some products are manually removed from the production system for different purposes, e.g., assessing if they meet quality requirements. This involves that incomplete traces may be included in $Traces(Sua)$, i.e., the traces that are not

[1] http://www.drools.org/

obtained from an entry point of the production system or that not end in one of its exit points. We consider that these traces neither express the system normal functioning nor express failures. Hence, it has been chosen to filter them out. This can been done by means of the parameter $point$, which stores the physical locations of devices.

The detection of the entry and exit points of Sua, denoted $Entry(Sua)$ and $Exit(Sua)$ is automatically performed with machine learning, and more precisely by using an outlier detection ([21]) on $Traces(Sua)$. An outlier is an observation that deviates so much from the other observations as to arouse suspicions that it was generated by a different mechanism. More precisely, we chose to use the *K-means clustering* method, a machine learning algorithm, which does not need to be trained before being effectively used (that is called unsupervised learning) and which is considered as one of the most robust and efficient clustering algorithm [41]. *K-means clustering* aims to partition N observations into K clusters. Here, observations are represented by the assignments of the variable $point$ present in each trace of $Traces(Sua)$, which captures device locations. To detect the entry points, we collect all the assignments found in the first production events of the traces. As we want to group the outliers together (the entry points), and leave the others in another cluster, we use $K = 2$. The same step is followed to get the exit points, except that we collect the assignment of the last events of the traces. This method is suitable for production systems because they hold physical assembly lines where almost all the products flow from real entry and exit points. Hence, on sufficiently large production event sets, we should observe high ratios of products moving from the same (entry) points to the same exit points. Using K-means helps automate this step, but, as explained in Section 7, when there are only few production events, we need an expert to provide them.

The devices may successively perform the same request several times with *repetitive patterns* of events. We have also chosen to remove the traces including repetitive patterns in order to reduce the final model size. The detection of repetitive patterns in traces is highly facilitated by firstly removing the variable assignments related to products since these assignments are always different among the traces. We call this step *normalisation*. For Michelin systems, the variables related to products are pid (product identifier) and $time$, which is assigned to a timestamp value. We define the operator $Norm_Y$, with Y the variable set related to products, to normalise traces. Autofunk tries to detect repetitive patterns in this way: if it finds a trace t of the form $t_1 p ... p t_2$ and another trace $t'_1 p' t'_2$ such that $Norm_Y(t_1) = Norm_Y(t'_1)$, $Norm_Y(t_2) = Norm_Y(t'_2)$ and $Norm_Y(p) = Norm_Y(p')$, then $Norm_Y(p)$ is designated as a repetitive pattern and t is removed from $Traces(Sua)$ since we suppose that t does not express a new and interesting behaviour.

Traces are completely removed rather than cleaning them by deleting the repetitive patterns to prevent from encoding behaviours not observed from Sua. An algorithm, which detects repetitive patterns is provided in [33]. Here, we prefer giving the definition of the *filtered trace* set, denoted $FTraces(Sua)$, derived from $Traces(Sua)$. We also define the trace normalisation with $Norm$ and denote the detected repetitive pattern set with $Pattern(Sua)$:

Definition 7 (Filtered traces) *Let $Traces(Sua)$ be the trace set of the production system under analysis $Sua = (Q, q_0, \Sigma, \rightarrow)$. $Entry(Sua)$, $Exit(Sua)$ denote its (physical) entry and exit point sets respectively. Let also $a_1(\alpha_1)...a_k(\alpha_k)$ be a trace of $Traces(Sua)$.*

1. *$P = \{x \mid \alpha_i (1 \leq i \leq k) \in D^X$ and $x \in X\}$. $Y \subseteq P$ is the set of variables related to products found in P;*
2. *$Norm_Y(a_1(\alpha_1)...a_k(\alpha_k)) =_{def} a_1(\alpha'_1)... a_n(\alpha'_k)$ with $\alpha'_i = \alpha_i(P \setminus Y)(1 \leq i \leq k)$;*
3. *$R = \{t_1 p ... p t_2 \in Traces(Sua) \mid t'_1 p' t'_2 \in Traces(Sua), Norm_Y(p_1) = Norm_Y(p'_1), Norm_Y(p_2) = Norm_Y(p'_2), Norm_Y(p) = Norm_Y(p')\}$;*
4. *$Pattern(Sua) =_{def} \{Norm_Y(t) \mid t_1 p ... p t_2 \in R\}$;*
5. *$FTraces(Sua) =_{def} \{a_1(\alpha_1)...a_k(\alpha_k) \in Traces(Sua) \mid \alpha_1(point) \in Entry(Sua)$ $\alpha_k(point) \in Exit(Sua)\} \setminus R$.*

The trace set $FTraces(Sua)$ only includes the traces capturing executions started from an entry point of Sua and ending to one of its exit points such that the traces do not contain several successive repetitive patterns of events. Traces in $FTraces(Sua)$ capture some behaviours of Sua but do not express more behaviours. Trace inclusion with $Traces(Sua)$ is preserved:

Proposition 8 $FTraces(Sua) \subseteq Traces(Sua)$.

With the example of production events given in Figure 4, we obtain the following filtered trace set: $FTraces(Sua) = \{17011(nsys := 1, nsec := 8, point := 1, pid := 1)\ 17021(nsys :=$

$1, nsec := 8, point := 3, tpoint := 8, pid := 1)$, $17011(nsys := 1, nsec := 8, point := 2, pid := 2)$ $17021(nsys := 1, nsec := 8, point := 4, tpoint := 9, pid := 2)\}$.

Finally, $FTraces(Sua)$ is partitioned into $ST = (ST_1, ..., ST_n)$ such that ST_i is a trace subset expressing behaviours starting at one entry point of $Entry(Sua)$ $(Card(Entry(Sua)) = n)$.

5.3 STS generation and reduction

Each trace subset ST_i of $ST = (ST_1, ..., ST_n)$ is then lifted to the level of the STS formalism, by transforming traces to runs and runs to STS paths. Given a set ST_i, its traces are converted into runs by completing them with states. Each run begins with the initial state $(l_0, v_\emptyset)$ with $v_\emptyset$ an empty condition. Then, new states are injected after each production event. The set of runs $Runs_i$ obtained from ST_i is defined as :

Definition 9 (Structured Runs) *Let ST_i be a trace set obtained from $FTraces(Sua)$. We denote $Runs_i$ the set of runs derived from ST_i with the following inference rule:*

$$\frac{t_{id}=(a_1,\alpha_1)...(a_k,\alpha_k)\in ST_i, \alpha_1(pid)=(pid:=id)}{(l0,v_\emptyset)a_1(\alpha_1)(l_{id1},v_\emptyset)...(l_{idk-1},v_\emptyset)a_k(\alpha_k)(l_{idk},v_\emptyset)\in Runs_i}$$

The runs of $Runs_i$ have states that are unique except for the initial state $(l0, v_\emptyset)$. We defined such a set to later build a STS having a tree structure. The state uniqueness is here guaranteed by means of the trace identifier pid, which is unique for each trace.

Runs are transformed into STS paths that are assembled together by means of a union. The resulting STS forms a tree compound of branches starting from the location $l0$. Parameters and guards are extracted from the assignments found in production events. In this step, the events of the runs are normalised with the $Norm$ operator to remove the parameter assignments related to products (assignment of the variables pid and $time$ in our context). We obtain more generalised STSs, which express the behaviours of Sua, independently of the manufactured products.

Definition 10 *Given a run set $Runs_i$ and Y the set of variables related to products, $\mathcal{S}_i = (L_{\mathcal{S}_i}, l0_{\mathcal{S}_i}, V_{\mathcal{S}_i}, V0_{\mathcal{S}_i}, I_{\mathcal{S}_i}, \Lambda_{\mathcal{S}_i}, \rightarrow_{\mathcal{S}_i})$ is the STS expressing the (generalised) behaviours found in $Runs_i$ such that:*

- $L_{\mathcal{S}_i} = \{l \mid \exists r \in Runs_i, (l, v_\emptyset)$ *is a state of* $r\}$,
- $l0_{\mathcal{S}_i} = l0$ *is the initial location such that* $\forall r \in Runs_i$, r *starts with* $(l0, v_\emptyset)$,
- $V_{\mathcal{S}_i} = \emptyset$, $V0_{\mathcal{S}_i} = v_\emptyset$,

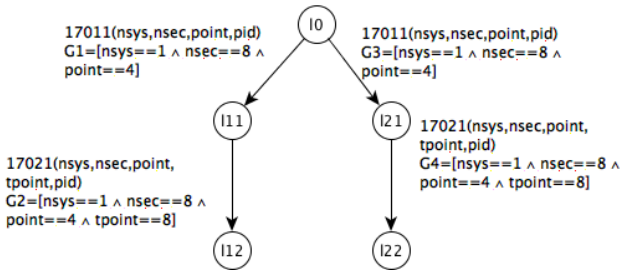

Figure 6: First inferred STS example

- $\rightarrow_{\mathcal{S}_i}$ *and* $\Lambda_{\mathcal{S}_i}$ *are defined by the following inference rule applied on every element* $r \in Runs_i$*:*

$$\frac{(l, v_\emptyset)a(\alpha)(l', v_\emptyset) \in r, a(\alpha') = Norm_Y(a(\alpha)), \quad p = \{x \mid \alpha \in D^X, x \in X\}, \quad G = \bigwedge_{(x:=v)\in\alpha'} x == v}{l \xrightarrow{a(p),G}_{\mathcal{S}_i} l'}$$

The STS $\mathcal{S}_i$ has a tree form, one branch transposing one trace of ST_i. Figure 6 illustrates the STS $\mathcal{S}_1$ obtained from the production events of Figure 4. It is composed of events, each associated with its own parameters. Transitions are labelled with guards directly derived from the parameter assignments found in the production events.

Once the subsets $ST_1, \ldots, ST_n$ are transformed into STSs, we obtain a first production system model $S = (\mathcal{S}_1, \ldots, \mathcal{S}_n)$. We also attach to S, the repetitive patterns, the entry and exit points detected from Sua: $Pattern(S) = Pattern(Sua)$, $Entry(S) = Entry(Sua)$, $Exit(S) = Exit(Sua)$.

If we only have collected a subset of all the production events of Sua, we have inferred a partial model S. In a sense, one could also affirm that S is an over-approximation of Sua because the pid variable can take any value. But S gives the same traces as Sua with the same parameter values, if the variables related to products are not considered. In this context, we say that S is not an over-approximation of Sua. This is captured by the following proposition:

Proposition 11 *Let S be a model inferred from a production system Sua and $Ld \subseteq \bigcup_{\mathcal{S}_i \in S} L_i$ be its deadlock locations. We have: $Norm_Y(FTraces(Sua)) = Norm_Y(Traces_{Ld}(S))$*

5.4 STS reduction

The model S includes STSs that are most likely too large for being employed for testing (e.g., for being

kept resident in memory). Yet, production systems are often conceived with a finite number of assembly lines and build products with a finite number of steps. A STS $\mathcal{S}_i$ in S should contain branches capturing the same sequences of events (without necessarily the same parameter assignments) and could be minimised.

Initially, we studied the state merging techniques used with passive inference methods, and particularly kTail, which appeared to be well suited in this context. But, as stated earlier, these techniques build over-approximated models by raising the level of abstraction, which may lead to false positives when used for testing. Consequently, we have chosen to design a context-specific and lightweight STS reduction technique, which aims at reducing a STS $\mathcal{S}_i$ into another STS denoted $R(\mathcal{S}_i)$. The reduction technique merges the STS branches having the same sequence of events. As we do not want to infer approximated models, when STS branches are merged, we keep all the guards and wrap them into matrices of guards. These matrices pack the data, which are separated from the transitions. This structure helps analyse events and parameters separately. In addition, it allows the recovery of the initial traces of $\mathcal{S}_i$. Finally, the resulting STS $R(\mathcal{S}_i)$ still keeps its tree structure but has fewer branches.

Given a STS $\mathcal{S}_i$, its paths are firstly adapted to express sequences of guards in a vector form. Later, the concatenation of these vectors shall give birth to matrices. This adaptation is obtained with the definition of the STS operator Mat:

Definition 12 *Let $\mathcal{S}_i = (L_{\mathcal{S}_i}, l0_{\mathcal{S}_i}, V_{\mathcal{S}_i}, V0_{\mathcal{S}_i}, I_{\mathcal{S}_i}, \Lambda_{\mathcal{S}_i}, \rightarrow_{\mathcal{S}_i})$ be a STS inferred from Sua. We denote $Mat(\mathcal{S}_i)$ the STS operator that consists in expressing guards of STS paths in a vector form.*

$Mat(\mathcal{S}_i) = (L_{Mat(\mathcal{S}_i)}, l0_{Mat(\mathcal{S}_i)}, V_{Mat(\mathcal{S}_i)}, V0_{Mat(\mathcal{S}_i)}, I_{Mat(\mathcal{S}_i)}, \Lambda_{Mat(\mathcal{S}_i)}, \rightarrow_{Mat(\mathcal{S}_i)})$ *where:*

- $L_{Mat(\mathcal{S}_i)} = L_{\mathcal{S}_i}$, $l0_{Mat(\mathcal{S}_i)} = l0_{\mathcal{S}_i}$, $I_{Mat(\mathcal{S}_i)} = I_{\mathcal{S}_i}$, $\Lambda_{Mat(\mathcal{S}_i)} = \Lambda_{\mathcal{S}_i}$,
- $V_{Mat(\mathcal{S}_i)}$, $V0_{Mat(\mathcal{S}_i)}$ *and* $\rightarrow_{Mat(\mathcal{S}_i)}$ *are given by the following rule:*

$$\frac{b{=}l_0 \xrightarrow{(a_1(p_1),G_1)...(a_k(p_k),G_k)}_{\mathcal{S}_i} l_k}{V0_{Mat(\mathcal{S}_i)} = V0_{Mat(\mathcal{S}_i)} \wedge M_b == [G_1,...,G_k] \quad l0_{Mat(\mathcal{S}_i)} \xrightarrow{(a_1(p_1),M_b[1])...(a_k(p_k),M_b[k])}_{Mat(\mathcal{S}_i)} l_k}$$

Given a path $b \in \rightarrow^*_{Mat(\mathcal{S}_i)}$, we also denote $Mat(b) = M_b$ the vector used within the guards of b.

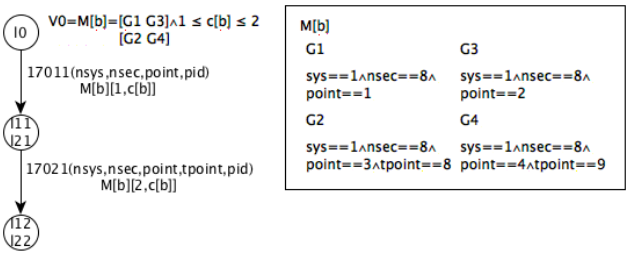

Figure 7: Reduced model (STS)

The STS branches having the same sequences of events can now be assembled. These branches are grouped into path equivalence classes:

Definition 13 (STS path equivalence class) *Let $\mathcal{S}_i = (L_{\mathcal{S}_i}, l0_{\mathcal{S}_i}, V_{\mathcal{S}_i}, V0_{\mathcal{S}_i}, I_{\mathcal{S}_i}, \Lambda_{\mathcal{S}_i}, \rightarrow_{\mathcal{S}_i})$ be a STS obtained from Sua.*

$[b]$ denotes the equivalence class gathering the paths of $Mat(\mathcal{S}_i)$ such that:

$[b] = \{b' = l0_{Mat(\mathcal{S}_i)} \xrightarrow{(a'_1(p'_1),G'_1),...,(a'_k(p'_k),G'_k)} l'_k \mid b = l0_{Mat(\mathcal{S}_i)} \xrightarrow{(a_1(p_1),G_1)...(a_k(p_k),G_k)} l_k, a_i(p_i) = a'_i(p'_i)\ (1 \leq i \leq k)\}$

The reduced STS $R(\mathcal{S}_i)$ of $\mathcal{S}_i$ is derived by merging the locations of the paths of every equivalence class $[b]$ found in $Mat(\mathcal{S}_i)$. The vectors of guards found in the paths of $[b]$ are joined into the matrix $M_{[b]}$. $R(\mathcal{S}_i)$ is defined as follows:

Definition 14 (STS reduction) *Let $\mathcal{S}_i = (L_{\mathcal{S}_i}, l0_{\mathcal{S}_i}, V_{\mathcal{S}_i}, V0_{\mathcal{S}_i}, I_{\mathcal{S}_i}, \Lambda_{\mathcal{S}_i}, \rightarrow_{\mathcal{S}_i})$ be a STS inferred from Sua. The reduction of $\mathcal{S}_i$ is modelled by the STS $R(\mathcal{S}_i) = (L_R, l0_R, V_R, V0_R, I_R, \Lambda_R, \rightarrow_R)$ where:*

$$[b] = \{b_1,...,b_m\} \text{ with } b_j = \frac{l0_{Mat(\mathcal{S}_i)} \xrightarrow{(a_1(p_1),G_{j1})...(a_k(p_k),G_{jk})}_{Mat(\mathcal{S}_i)} l_{jk}}{V0_R = V0_R \wedge M_{[b]} == [Mat(b_1),...,Mat(b_m)] \wedge (1 \leq c_{[b]} \leq m), l0_R \xrightarrow{(a_1(p_1),M_{[b]}[1,c_{[b]}])...(a_k(p_k),M_{[b]}[k,c_{[b]}])}_R (l_{1k}...l_{mk})}$$

A STS $R(\mathcal{S}_i)$ holds paths labelled by guards, which refer to a matrix $k \times m$ denoted $M_{[b]}$. A column of this matrix $M_{[b]}$ stores a list of successive guards found in a path of the initial STS $\mathcal{S}_i$. The choice of the column in a matrix depends on a new variable $c_{[b]}$, which takes a value between 1 and m by means of the initial condition $V0_R$.

The STS example of Figure 6 has two paths having the same sequence of events, which can be reduced to one path. The reduced STS is depicted in Figure 7. The guards are placed into two vectors

$M1 = [G1\ G2]$ and $M2 = [G3\ G4]$. These are combined into the matrix $M_{[b]}$. The condition $V0$ exhibits that the variable $c_{[b]}$ can either be assigned to $c_{[b]} := 1$ or $c_{[b]} := 2$, in reference with the two matrix columns. The resulting STS of Figure 7 still encodes the initial behaviours described by the first STS. This is captured with the following proposition:

Proposition 15 *Let $\mathcal{S}_i$ be a STS inferred from Sua and $R(\mathcal{S}_i)$ be its reduced STS.*
We have: $Traces(R(\mathcal{S}_i)) = Traces(\mathcal{S}_i)$.

When we apply this STS reduction method on the whole production system model S, we obtain a model denoted $R(S)$ such that $R(S) = (R(\mathcal{S}_1), \ldots, R(\mathcal{S}_n))$, and $Entry(R(S)) = Entry(S)$, $Exit(R(S)) = Exit(S)$, $Pattern(R(S)) = Pattern(S)$.

5.5 Soundness complexity and termination of the model inference step

The soundness of the model inference stage mainly depends on the inference rules and the knowledge bases. The latter are exclusively constituted of (positive) facts (Events, Traces, Transitions, STSs) that have an Horn form. We have assumed that the inference rules are Modus Ponens. Therefore, the resolution of the inference rules and the inference of knowledge bases is sound and complete [9] (all the resulting facts are always true and have proofs in the system). From this fact and by considering the propositions 8, 11 and 15, we can state the soundness of the whole model inference stage with:

Proposition 16 *Let Sua be a system under analysis, which meets the assumptions given in Section 3. Let $S = (R(\mathcal{S}_1), \ldots R(\mathcal{S}_n))$ be the production system model inferred from Sua and $Ld \subseteq \bigcup_{1 \leq i \leq n} L_{R(\mathcal{S}_i)}$ be its deadlock location set. We have:*

1. $FTraces(Sua) \subseteq Traces(Sua)$ *and* $FTraces(Sua) \subseteq Traces_{Ld}(R(S))$
2. $Norm_Y(Traces_{Ld}(R(S))) = Norm_Y(FTraces(Sua)) \subseteq Norm_Y(Traces(Sua))$

The model inference complexity is polynomial in time and is proportional to $O(t + m(t^2 + t + k + log(m)))$ with m production events, t traces in $Traces(Sua)$, k inference rules for filtering (worst case). The complexity to filter m production events with k rules is $O(mk)$. The remaining ones are sorted in $O(m * log(m))$ (with the Java $Collection.sort()$). We mine $Traces(Sua)$ with K-means, whose complexity is proportional to $O(2t)$ with t the number of traces in $Traces(Sua)$. The complexity of extracting $FTraces(Sua)$ from $Traces(Sua)$ in $O(t + t^2m)$. Indeed, inference rules covers the traces of $Traces(Sua)$ to keep those capturing behaviours starting from one entry point and finishing at one exit point of Sua. For every pair of traces, we search for repetitive patterns by scanning the traces labels (with the Michelin context, we limit to the patterns of the form request/response) and then check if the parameter values are identical. Traces are lifted to the STS level, event after event, with a complexity proportional to $O(m)$. The STS reduction complexity is $O(m + tm)$. Path equivalence classes are generated with a hash function, called on event sequences (complexity proportional to $O(m)$). The paths of a class are grouped with a rule covering all the paths together $(O(tm))$.

6 Passive testing

In this section, we present the second part of our framework, dedicated to the passive testing of another system under test Sut. The model $R(S)$, inferred from a system Sua, is regarded as a reference specification. The STSs of $R(S)$ can be modified to record some updates made on Sut or some specific features. Manual modifications of the inferred STSs are possible but should be time-consuming though, since the STSs are usually large. Inference rules added by experts are another possibility to update the STSs of $R(S)$, e.g., with variable or label modifications.

In addition, we take advantage of this step to label the final locations of the STSs of $R(S)$ with "Pass". We denote these locations as verdict locations and gather them into the set $Pass \subseteq \bigcup_{1 \leq i \leq n} L_{R(\mathcal{S}_i)}$. For sake of readability, we still denote the resulting model with $R(S)$. It expresses some possible behaviours that should happen, which are encoded by the traces $Traces_{Pass}(R(S))$. We refer to these traces as pass traces. We call the others, possibly failure traces because $R(S)$ is a partial model.

To reason about conformance between a specification and an implementation, it is classically assumed that the latter can be modelled with an unknown and possibly non-deterministic model. In our context, we suppose that a production system under test can be modelled by a LTS denoted Sut.

An overview of the passive testing module of Autofunk is depicted in Figure 8. Testing is passively performed, i.e., a set of production events were collected before from Sut, in the same way as for Sua. These are grouped into traces to form the trace set $Traces(Sut)$. Traces are then filtered as described

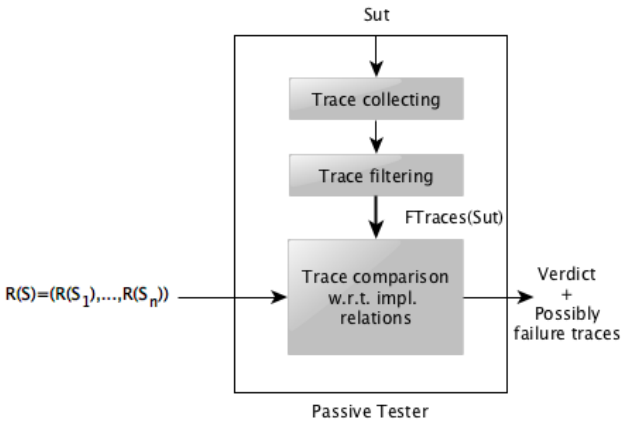

Figure 8: Passive Testing in Autofunk

in Section 5.1 to obtain a set of filtered traces denoted with $FTraces(Sut)$. The tester is finally called to check if Sut is conforming to $R(S)$.

6.1 Implementation relations

We define conformance with two implementation relations, which take root in the preorder relations ([14]). Our industrial partner was interested in checking whether the filtered traces of Sut match the behaviours encoded in $R(S)$. In this case, the test verdict must reflect a successful result. Conversely, if a filtered trace of Sut cannot be found in the traces of $R(S)$, it remains difficult to conclude that Sut is a faulty implementation because $R(S)$ is a partial model. It does not necessarily encode all the correct behaviours of the previous system under analysis Sua. This is why we consider such a trace as a possibly failure trace.

This concept of conformance is defined by a first implementation relation, denoted with $\leq_{ft}$ (ft for filtered traces). It aims at checking whether the filtered traces of Sut are Pass traces of $R(S)$:

Definition 17 (Implementation relation $\leq_{ft}$) *Let $R(S)$ be an inferred model of Sua and Sut be the system under test.When Sut produces filtered traces also captured within $R(S)$, we write: $Sut \leq_{ft} R(S) =_{def} FTraces(Sut) \subseteq Traces_{Pass}(R(S))$.*

As the model $R(S)$ can be under-approximated and may not include some correct behaviours, Michelin engineers wished the definition of a second relation "less strict on the parameters on condition that these parameters could be found inside the model". In a sense, one can say that this second relation aims at describing an over-approximation of the under-approximation, which contradicts the objective of the model inference stage. This implementation relation must be considered as a complementary relation of $\leq_{ft}$, which is fully useful when $\leq_{ft}$ returns a possibly failure trace of Sut. The second relation aims to point out whether this trace might reflect a realistic scenario, composed of a correct sequence of events completed of parameters found in other traces of the model $R(S)$. This helps classify possibly failure traces in risk importance. In the Michelin context, when a trace of Sut is a possibly failure trace for both relations, the likelihood of failure detection is the highest.

This relation, denoted $\leq_{mft}$ (with mft for multiple filtered traces), defines that an implementation Sut is correct iff its filtered traces $a_1(\alpha_1)...a_k(\alpha_k)$ can be found in several traces of $Traces_{Pass}(R(S))$ having the same sequence of labels $a_1...a_k$. The implementation relation $\leq_{mft}$, is defined by:

Definition 18 (Implementation relation $\leq_{mft}$) *Let $R(S) = (R(\mathcal{S}_1), \ldots, R(\mathcal{S}_n))$ be an inferred model of Sua, Sut be the system under test and Y be the set of variables related to products.*

$Sut \leq_{mft} R(S) =_{def} \forall t = a_1(\alpha_1)...a_k(\alpha_k) \in FTraces(Sut), \forall \alpha_j(x)_{(1\leq j\leq k)}$ *with* $x \notin Y$, $\exists t' \in Traces_{Pass}(R(S)) : t' = a_1(\alpha'_1)...a_k(\alpha'_k), \alpha'_j(x) = \alpha_j(x)$

If we take back the example of Figure 7, the trace $t = 17011(nsys := 1, nsec := 8, point := 1, pid := 1)\ 17021(nsys := 1, nsec := 8, point := 4, tpoint := 9, pid := 1)$ is not a pass trace of $R(S)$ for the relation $\leq_{ft}$ because this trace cannot be extracted from the STS of Figure 7, even though it has the same sequence of labels "17011 17021" as t. If we focus on the guards and on the matrix $M_{[b]}$, the parameter assignments of the event $17011(nsys := 1, nsec := 8, point := 1)$ satisfy the guard G1 but those in $17021(nsys := 1, nsec := 8, point := 4, tpoint := 9)$ do not meet the guard G2 on account of the variables $point$ and $tpoint$, which do not take the expected values. In the second column of the matrix, the first guard $G3$ does not hold with $(nsys := 1, nsec := 8, point := 1)$. With the implementation relation $\leq_{mft}$, each parameter assignment $\alpha_j(x)$ found in the trace t (except those related to products), must also be found in one of the traces of the STS of Figure 7 having the same sequence of labels as t. In the STS level, this involves that any parameter assignment $\alpha_j(x)$ found in the trace t must meet at least one guard of the matrix line j in $M_{[b]}[j, *]$. The parameter assignments of the event $17011(nsys := 1, nsec := 8, point := 1)$ satisfy the guard G1 of the STS Figure 7 and those in

$17021(nsys := 1, nsec := 8, point := 4, tpoint := 9)$ satisfy the guard G4. Hence, the trace t is pass trace for the relation $\leq_{mft}$.

By generalising this idea, we propose to rewrite the implementation relation $\leq_{mft}$ in an equivalent but simpler form in the following. Firstly, $\leq_{mft}$ can be written with the guards of the STSs of $R(S)$:

Proposition 19 $Sut \leq_{mft} R(S)$ *iff* $\forall t = a_1(\alpha_1)...a_k(\alpha_k) \in FTraces(Sut), \exists (1 \leq i \leq n), \exists b = l0_{R(\mathcal{S}_i)} \xrightarrow{(a_1(p_1),M_{[b]}[1,c_{[b]}]),...,(a_k(p_k),M_{[b]}[k,c_{[b]}])} Pass$ *with* $M_{[b]}$ *a matrix* $k \times c$, $\forall \alpha_j(x)(1 \leq j \leq k) : \alpha_j(x) \models Proj_{\{x\}}(M_{[b]}[j,1] \vee ... \vee M_{[b]}[j,c])$.

Now, one can compare both relations $\leq_{ft}$ and $\leq_{mft}$. With the relation $\leq_{ft}$, the variable assignments found in a trace of Sut must meet the guards of one matrix column of $R(\mathcal{S}_i)$ (any column but always the same for all the assignments of the trace). With $\leq_{mft}$, the variable assignments of the trace of Sut must meet the guards of any column. This attests that $\leq_{mft}$ is a weaker relation than $\leq_{ft}$ and that:

Proposition 20 $Sut \leq_{ft} R(S) \implies Sut \leq_{mft} R(S)$

The sketch of proof of Proposition 20 is given in [34]. The relation $\leq_{mft}$ can be simplified again by adapting the disjunction of guards $M_{[b]}[j,1] \vee ... \vee M_{[b]}[j,c]$, found in the matrix $M_{[b]}$. The resulting formula can be reduced by gathering all the equalities $(x == val)$ together with disjunctions for each variable x. We obtain one guard of the form $\bigwedge_{x \in I}(x == val_1 \vee ... \vee x == val_k)$. If we generalise this idea on the matrices of a STS $R(\mathcal{S}_i)$, it becomes possible to replace a matrix composed of several columns into one vector of guards. Therefore, we propose to transform the STS $R(\mathcal{S}_i)$ into another STS composed of such vectors.

Definition 21 *Let* $R(\mathcal{S}_i) = (L_R, l0_R, V_R, V0_R, I_R, \Lambda_R, \rightarrow_R)$ *be a reduced STS inferred from* Sua. *We denote* $D(\mathcal{S}_i)$ *the STS* $(L_D, l0_D, V_D, V0_D, I_D, \Lambda_D, \rightarrow_D)$ *derived from* $R(\mathcal{S}_i)$ *such that:*

- $L_D = L_R$, $l0_D = l0_R$, $I_D = I_R$, $\Lambda_D = \Lambda_R$,
- $V_D, V0_D$ *and* $\rightarrow_D$ *are given by the following inference rule:*

$$\frac{b = l0_R \xrightarrow{(a_1(p_1),M'_{[b]}[1,c'_{[b]}])...(a_k(p_k),M'_{[b]}[k,c'_{[b]}])} {}_R\ l_k \quad (1 \leq c'_{[b]} \leq c)\ in\ V0_R}{l0_D \xrightarrow{(a_1(p_1),M_{[b]}[1])...(a_k(p_k),M_{[b]}[k])} {}_D\ l_k}$$
$$V0_D = V0_D \wedge (c_{[b]} == 1) \wedge M_{[b]},$$
$$M_{[b]}[j]_{(1 \leq j \leq k)} = \bigwedge_{x \in p_j} (Proj_{\{x\}}(M'_{[b]}[j,1]) \vee ... \vee Proj_{\{x\}}(M'_{[b]}[j,c]))$$

When we apply this transformation on the STSs of $R(S) = (R(\mathcal{S}_1), \ldots, R(\mathcal{S}_n))$, we obtain a model denoted $D(S) = (D(\mathcal{S}_1), \ldots, D(\mathcal{S}_n))$.

The second implementation relation $\leq_{mft}$ can now be expressed with $D(S)$:

Proposition 22 $Sut \leq_{mft} R(S)$ *iff* $\forall t = a_1(\alpha_1)... a_k(\alpha_k) \in FTraces(Sut), \exists (1 \leq i \leq n), \exists l0_{D(\mathcal{S}_i)} \xrightarrow{(a_1(p_1),G_1),...,(a_k(p_k),G_k)} Pass$ *such that* $\forall \alpha_j (1 \leq j \leq k), \alpha_j \models G_j$.

$\leq_{mft}$ now implies that a correct trace of Sut is also a pass trace of the model $D(S)$. This notion of trace inclusion is close to the idea formulated with the first relation $\leq_{ft}$, and finally the relation $\leq_{mft}$ can be written with $\leq_{ft}$ as:

Proposition 23

1. $Sut \leq_{mft} R(S)$ *iff* $FTraces(Sut) \subseteq \bigcup_{1 \leq i \leq n} Traces_{Pass}(D(\mathcal{S}_i))$.
2. $Sut \leq_{mft} R(S) \Leftrightarrow Sut \leq_{ft} D(S)$.

This proposition implies that the same passive tester algorithm can be called to check if both relations hold for a system under test.

In addition to these two relations, others could be implemented, e.g., ioco [36] or invariant satisfiability checking [10], etc. In the Michelin context, the two previous relations where considered appropriate for the testing purposes of the company.

6.2 Passive tester algorithm

The passive tester functioning is deducible from the above propositions. Indeed, it takes the models $R(S)$ and $D(S)$ and checks whether the filtered traces of $FTraces(Sut)$ belong to $Traces_{Pass}(R(S))$ or $Traces_{Pass}(D(S))$ respectively.

The passive tester algorithm is presented in Algorithm 1. It takes the traces of Sut and the

model $R(S)$. It starts by constructing the filtered traces $FTraces(Sut)$ from $Traces(Sut)$ and builds $D(S)$. Afterwards (line 4), it covers every trace t of $FTraces(Sut)$ and tries to find a STS $R(\mathcal{S}_i)$ such that t is a pass trace in $Traces_{Pass}$ $(R(\mathcal{S}_i))$. This step is performed with the function $complies\_with(tracet, STS\ S)$. If this function returns True, it is not necessary to check if the traces t satisfies the second relation $\leq_{mft}$ since $\leq_{ft} \implies \leq_{mft}$ (Proposition 20). On the contrary, the trace t is placed into the set T_1, which gathers the possibly failure traces w.r.t. $\leq_{ft}$. The algorithm performs the previous step once more but on $D(S)$. As previously, if the function returns False, the trace t is placed into the set T_2. The latter gathers the possibly failure traces, w.r.t. the relation $\leq_{mft}$. Finally, if T_1 is empty, the verdict "Pass$\leq_{ft}$" is returned, which means that the first implementation relation holds. Otherwise, T_1 is provided. If T_2 is empty, the verdict "Pass$\leq_{mft}$" is returned, or T_2 in the other case.

The function *complies_with(trace t, STS S)* (lines 22-32) aims at checking whether the trace $t = a_1(\alpha_1)...a_k(\alpha_k)$ is a trace of the STS S. More precisely, if a STS path b is composed of the same sequence of labels as the trace t, the function tries to find a matrix column $C = M_{[b]}[*, i]$ such that every guard $C[j]$ $(1 \leq j \leq k)$ holds with the variable assignment α_j. If such a column of guards exists, the function returns True or False otherwise.

When one of the implementation relations does not hold, this algorithm offers the advantage to provide the possibly failure traces of $FTraces(Sut)$. Such traces can later be analysed.

Algorithm 1: Passive testing algorithm

```
   input : R(S), Traces(Sut)
   output: Verdicts and possibly failure trace sets T1, T2
1  T1 = T2 = ∅;
2  Build FTraces(Sut);
3  Build D(S);
4  foreach t ∈ FTraces(Sut) do
5      found=false;
6      foreach i ∈ 1, ..., n do
7          if complies_with(t, R(Si)) then
8              found=true; break;
9      if found == false then
10         T1 = T1 ∪ {t};
11         foreach i ∈ 1, ..., n do
12             if complies_with(t, D(Si)) then
13                 found=true; break;
14         if found == false then
15             T2 = T2 ∪ {t}
16 if T1 == ∅ and T2 == ∅ then
17     return "Pass≤ft, Pass≤mft"
18 if T1 ≠ ∅ and T2 == ∅ then
19     return T1, "Pass≤mft"
20 if T1 ≠ ∅ and T2 ≠ ∅ then
21     return T1, T2

22 Function complies_with(trace t, STS S ) : bool is
23     if ∃b = l0S --(a1(p1),G1)...(ak(pk),Gk)--> Pass : trace =
       a1()α1) ... ak(αk) then
24         M[b] = Mat(b) is the Matrix l × c of b;
25         i = 1;
26         while i ≤ c do
27             C = M[b][*, i];
28             foreach j ∈ 1, ..., k do
29                 if αj ⊭ C[j] then break ;
30             if j == k then return True ;
31             i++;
32     return False;
```

6.3 Soundness, termination and complexity of Algorithm 1

The soundness of Algorithm 1 is captured by the following proposition, whose sketch of proof is given in [34]:

Proposition 24 *1. $Sut \leq_{ft} R(S) \implies$ Algorithm 1 returns "Pass$\leq_{ft}$", "Pass$\leq_{mft}$"*

2. $Sut \leq_{mft} R(S) \implies$ Algorithm 1 returns "Pass$\leq_{mft}$"

Algorithm 1 terminates if $FTraces(Sut)$ is finite (which is necessarily the case since the filtered traces are built from a finite number of production events). For every trace in $FTraces(Sut)$, it covers the paths of the STSs of $R(S)$ and $D(S)$. The STSs in $(R(\mathcal{S}_1), \ldots, R(\mathcal{S}_n))$ and in $(D(\mathcal{S}_1), \ldots, D(\mathcal{S}_n))$ have a finite number of paths (at worst equal to the number of traces in $FTraces(Sua)$).

The complexity of Algorithm 1 is proportional to $O(mk + mlog(m) + t + t^2m + tmc))$ with t the number of filtered traces of Sut, m the number of production events, k the number of inference rules and c the highest number of columns in any matrix $M_{[b]}$. As in the model inference stage, it begins constructing $FTraces(Sut)$.This step has a complexity proportional to $O(mk + mlog(m) + t + t^2m)$. Then, Algorithm 1 calls the procedure $complies\_with$ twice for each trace in $FTraces(Sut)$. The complexity of the procedure $complies\_with$ is $O(m \times c)$. Indeed, finding a path in a STS related to a trace is negligible by using a hash mechanism to identify sequences of labels, hence, we only take the matrix traversal into account here.

7 Evaluation

The tool Autofunk is currently used with factory simulations and is deployed in one factory for testing purposes. It is planned to gradually deploy it in other ones. The tool was experimented in this factory to evaluate the following criteria:

- *C1 (Accuracy/precision):* are models accurate w.r.t a real production system? Do we always obtain the same results? Does the tool return Pass verdicts when the system under test is identical to the one used to infer models?
- *C2 (Efficiency/Effectiveness):* can a production system be tested in reasonable time? Does Autofunk detect faults? Does it help reduce the testing phase complexity?
- *C3 (Scalability):* can Autofunk take as inputs large trace sets and still build models and test systems in reasonable time?

Initially, while the Autofunk implementation, we executed functional tests in a controlled environment to check how Autofunk builds models and if it can detect faults. In a sense, these functional tests partially evaluate the criteria C1 and C2. Sua is materialised by a set of twenty traces constructed manually and composed of ten production events. The functional tests aimed to check that Autofunk always provides the expected STSs. Then, other functional tests were used to check that Autofunk detects faults. To materialise Sut, we kept the same trace set on which we injected faults. We considered the removal/addition of events, of parameters and of repetitive patterns. The functional tests seeded the tester module of Autofunk with this trace set and checked that the faults were detected. This preliminary phase does not completely answer to the previous questions though. This is why Autofunk was installed inside a real production system (on a Linux machine with 12 Intel(R) Xeon(R) CPU X5660 @ 2.8GHz and 64GB RAM) to assess under what real circumstances these criteria are fulfilled. This system is a workshop (part of a whole factory), in which two main operations are performed (with machines and human operators): tire assembling (assembling the components onto a tire building drum) and curing (applying pressure to the tire in a mold). It is composed of 3 entry points starting 3 main assembly lines split into a large set of sub-lines to reach devices and operators. The tires can be placed in storage areas in which they may stay several days up to some weeks, or go out of the workshop through 3 exit points. For confidentiality reasons, we cannot provide more details about the system neither a plan of its layout.

7.1 Empirical setup and results

We collected production events from this system by considering several (collection) delays to build models: 1, 8 11, 20 and 23 days.

Table 1 summarises the observed results. The third column gives the number of production events recorded on the system. The next column shows the trace number obtained after the parsing step. N and M represent the number of entry and exit points. The column *Trace Subsets* shows how $FTraces(Sua)$ is segmented per the entry point into subsets and the number of traces included in each subset. For instance, in the second experiment, three entry points are detected, hence, $FTraces(Sua)$ is partitioned into three subsets, which give birth to the same number of STSs. The trace numbers, given in this column, also correspond to the numbers of paths generated in the first STSs. The eighth column, # $R(\mathcal{S}_i)$, represents the number of paths found in each reduced STSs $R(\mathcal{S}_1), ..., R(\mathcal{S}_n)$, with n the number of entry points. Finally, execution times are rounded and expressed in minutes in the last column.

Based on the recommendations given by the Michelin engineers, we chose to keep as reference model Sua the one obtained in Experiment C. It expresses the system functioning, manufacturing the same product (same tire reference) and it includes the correct number of entry and exit points of the real system. This model was then used to passively test a modified version of the production system in order to detect regressions.

We collected three trace sets for testing at different periods of time to mostly cover all the cases that may be encountered: no modification of the production system, slight modifications (changes of some parameters, slight modifications of the devices, which is a typical case at Michelin), strong modifications of the original system or test of a different system. These experiments are presented in Table 2. The second column shows different kinds of system under test: $Sut1$ that was the same as Sua, $Sut2$ that was slightly updated from Sua with new application versions, and $Sut3$ that was a system much older than Sua and hence very different (older application versions, perhaps other devices, etc.). Column 3 gives the sizes of the trace sets used to infer models, column 4 the sizes of the trace sets collected from the systems under test. The two next columns show the percentage of pass traces w.r.t. the relations $\leq_{ft}$ and $\leq_{mft}$. The last column indicates the execution time for the testing phase.

Exp.	# Days	# Events	$Card(Traces(Sua))$	N	M	# FTraces Subsets	# $R(\mathcal{S}_i)$	Time (min)
A_1	1	660,431	16,602	2	3	4,822	332	1
A_2						1,310	193	
B_1	8	3,952,906	66,880	3	3	28,555	914	9
B_2						18,900	788	
B_3						6,681	51	
C_1	11	3,615,215	61,125	3	3	28,302	889	9
C_2						14,605	681	
C_2						7,824	80	
D_1	11	3,851,264	73,364	2	3	35,541	924	9
D_2						17,402	837	
E_1	20	7,635,494	134,908	2	3	61,795	1,441	16
E_2						35,799	1,401	
F_1	23	9,231,160	161,035	2	3	77,058	1,587	24
F_2						43,536	1,585	

Table 1: Results of 6 experiments on model inference

Exp.	Sut	$Card(Traces(Sua))$	$Card(FTraces(Sut))$	Pass$\leq_{ft}$	Pass$\leq_{mft}$	Time (min)
1	$Sut1 = Sua$	61,125	61,125	100%	100%	17
2	$Sut2$ updated from Sua	61,125	25,047	98%	98%	10
3	$Sut3 \neq Sua$	61,125	2,075	3%	30%	4

Table 2: Results of 3 experiments on passive testing

7.2 Evaluation

7.2.1 C1 (Accuracy/precision)

To answer the questions concerning these criteria, we firstly focused on model generation. We extracted the values of columns 4 and 7 in Table 1 to depict the stacked bar chart illustrated in Figure 9. This chart shows, for each experiment, the proportion of filtered traces kept to build models, over the initial number of traces in $Traces(Sua)$. The first limitation of Autofunk takes effect in Experiment A. Indeed, only 37% of the initial traces are kept to build models. There are two reasons for this. In one day, too few traces are gathered for using K-means with success. In this trace set, there is not a clear separation between the entry/exit points and the others. Therefore, K-means may return wrong point clusters. In this situation, entry and exit points have to be manually given by an engineer. This is what happened in Experiment A. The second reason concerns the collection delay itself. During a day, most of the recorded traces do not start or end at real entry or exit points of the production system, but rather start or end somewhere within assembly lines. Indeed, the workshop contains storage areas where products can stay for a while, depending on the production campaigns or needs for instance. That is why, on a single day, so many incomplete traces are filtered. With more production events, such a phenomenon is limited because these storage delays are absorbed in the period of time considered to collect production events. With the other experiments, the ratios of traces removed from the initial set $Traces(Sua)$ vary between 20 % to 30 %. After some inspections, we observed that, among these traces, around 15 % to 25% appear to be traces composed of repetitive patterns of events. The other traces capture abnormal behaviours of the production system, e.g., unexpected removal of products, device interruptions, etc. and are deleted according to the clusters of entry / exit points given by K-means. We observed that these ratios are acceptable, and few traces expressing normal behaviours of the production system are deleted. But the trace filtering could still be refined with more inference rules or with another cluster analysis technique. For instance, Table 1 revealed strange behaviours not taken into consideration by Autofunk. In experiments B and C, three entry points are detected whereas two are found in the others. Actually, the real production system has three entry points whose two are mainly used. The third one is employed to equilibrate the production load between this system and a second one located close to it in the same factory. Depending on the period, this entry point may be more or less solicited, hence the difference between experiments B, C and experiment D.

The inferred models from $FTraces(Sua)$ are accurate in the sense that the normalised traces of the models are equal to the normalised traces of $Traces(Sua)$ (Proposition 16). Regarding the test-

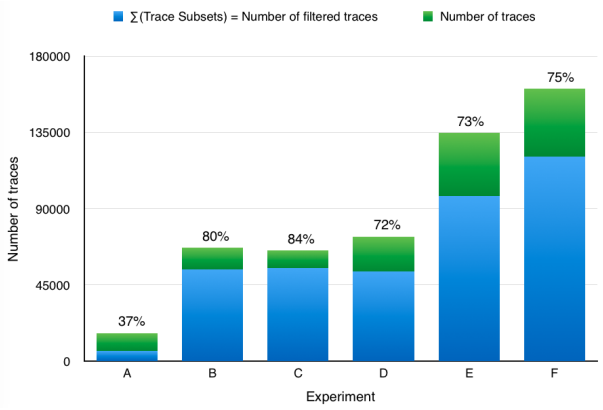

Figure 9: Proportions of filtered traces

ing phase, we measured accuracy with the first experiment of Table 2. The system under test is here exactly identical to the original production system Sua. This experiment shows that the passive tester yields the expected test verdicts, hence no fault were detected (no regression). Indeed, both relations $\leq_{ft}$ and $\leq_{mft}$ are satisfied. Finally, for a given trace set, if the traces are analysed in a different order, we still obtain the same results, because traces and STS paths are analysed/compared separately by the algorithms of Autofunk. K-means also computes the same clusters from the same trace set.

We can conclude that the accuracy and precision of Autofunk depends on the amount of traces collected to build models and on the delay considered to get these traces. The larger the trace set is and the longer the production events are collected, the more accurate the models are without any loss of precision. Otherwise, the main danger is that K-means might return wrong sets of entry/exit points, which have to be manually given. The best delay for collecting production events depends on the system behaviour. With the production system taken for experimentation, this delay has been set against the storage area functioning. We observed that if events are collected during 7 days or more, we obtain accurate and precise results.

7.2.2 C2 (Efficiency / Effectiveness)

One of the purposes of Autofunk is to automate the testing stage and to quickly return the possibly failure traces when non conformance is detected so that they can be later analysed by engineers to diagnose the cause of the non conformance detection (failure, correct behaviour not found in the model, new behaviour obtained after system updates).

Tables 1 and 2 show that Autofunk builds models and gives test verdicts in reasonable time. With production event sets collected during one day up to one week (experiments A, B, C, and D), models are inferred in less than 10 minutes. Experiment C in Table 1 corresponds to a typical use of Autofunk for Michelin. Models are generated after 9 minutes from production events collected during 11 days (more than three millions of events).

In Table 1, the difference of STS path numbers between the columns 7 and 8 clearly shows that our STS reduction approach is efficient. For instance, with experiment C, we reduce the STSs by 96.7%. In other words, 96% of the original behaviours are packed into matrices. These results mainly stem from our choice to design a context-specific state merging process. It is manifest that these ratios should vary with other kinds of systems. Experiment 2 in Table 2 also corresponds to a typical use of Autofunk for testing. The model were inferred from traces collected during 11 days. Traces of the system under test $Sut2$ were collected during 5 days. It took only 10 minutes to check whether Sut is conforming to the STSs inferred from Sua with respect to $\leq_{ft}$ and $\leq_{mft}$.

Experiment 2 in Table 2 shows that Autofunk is effective to detect faults: $Sut2$ was slightly updated from Sua and the tool detected that 98% of the traces are pass traces, the remaining 2% are new behaviours that never occurred before. Here, engineers have to manually analyse the possibly failure traces to check if these are the consequence of system failures or of new features of $Sut2$ not present in Sua. 2% means 500 traces, which remains a significant trace amount to inspect. Nonetheless in our manufacturing context, this is still valuable. Before Autofunk, engineers had to manually test the whole system by hand: around several hundreds of scenarios were executed and the resulting traces also had to be manually inspected. Autofunk can test a production system by automatically inspecting thousands of its traces in some minutes. It analyses more the system under test than manual testing, it remains for engineers to manually check a small subset of traces (only those capturing behaviours not found in the model). Such information is essential for Michelin engineers so that they can quickly focus on the potential defects of the system under test. In this example, we observed that almost half of the possibly failure traces were caused by unexpected manual interventions done by operators (tires were taken from one position and placed elsewhere). Around a hundred of traces captured new behaviours of $Sut2$ not possible in Sua. None of them revealed real failures in this case.

The third experiment illustrates an abnormal use of our framework. The defacto usage of our framework is to build models from a production system Sua, which should be older than an new or updated

system Sut. Here, the traces of $Sut3$ were collected long before collecting those of Sua used for inferring models. $Sut3$ were indeed a four month older system whose CIM2 applications were different than those of Sua. In this situation, Sua and $Sut3$ can be seen as quite distinctive production systems. Autofunk still detects non-conformance and provides possibly failure traces quickly. Unsurprisingly, both implementation relations are unsatisfied. Only 3% of the traces of $Sut3$ are pass traces w.r.t. $\leq_{ft}$. This means that only 60 traces of $Sut3$ exactly match the behaviours captured by the inferred models. With the second implementation relation $\leq_{mft}$, the pass trace ratio is increased to 30%. The second relation shows that roughly a third of the traces of $Sut3$ have the same sequences of events as the traces found in the STSs, but the parameter values (which can be found in other traces) are different. Hence, the second relation shows that 27% of the pass traces appear to be realistic scenarios. This helps focus on the traces having unknown sequences of actions or actions associated with unknown parameters, for which the likelihood of failure detection is the highest. Indeed, Michelin engineers confirmed that these traces often capture system defects.

These experiments revealed that Autofunk is efficient since models and possibly failure traces are provided in reasonable time. It can detect non-conformance when the system under test includes behaviours not present in the model and help focus on the unknown behaviours detected from Sut. However, when Sua and Sut have many differences, it detects non-conformance but returns so much possibly failure traces that it may become difficult to inspect all of them.

C3 (scalability)

The motivations behind this work and collaboration are to generate models for testing from large sets of production events and to do this as quick as possible so that models may be also used for other purposes, e.g., to diagnose unexpected stops or failures in the production system. The results given in Tables 1 and 2 reveal that our framework can take up to millions of production events and still build models quickly (less than half an hour). Experiment F handled almost 10 millions of events in less than half an hour to build two STSs including around 1,600 paths. As mentioned in Section **??**, the parsing step is not parallelised yet, and it took up to 20 minutes to open and parse around 1,000 files (number of Michelin log files for this experiment). This is a technical issue that needs to be addressed in the future.

The columns 3 and 8 of Table 1 are confronted in the graph of Figure 10 to summarise the performances of our framework, and how fast it infers models (experiments B, C and D run in about 9 minutes). Likewise, the columns 4 and 7 of Table 2 give the graph of Figure 11. The linear regressions depicted in these figures reveal that the overall framework scales well despite the current production event parsing implementation, by means of the parallelisation of the Autofunk algorithms (STS generation and reduction, trace comparison of the tester).

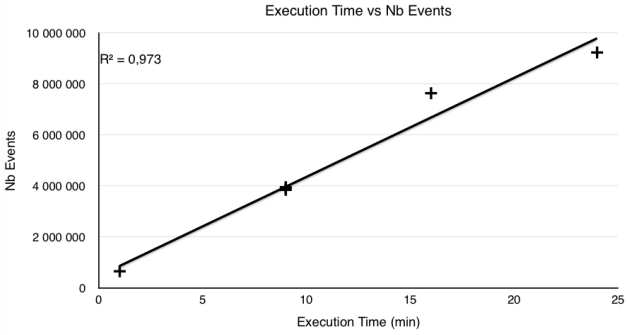

Figure 10: Model inference: Execution time vs Nb events

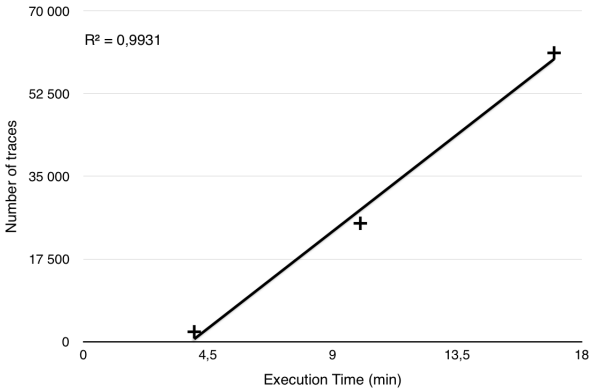

Figure 11: Passive testing: Execution time vs Nb events

The memory consumption peak occurs in Autofunk in the beginning of the model inference stage. Every production event is currently loaded in memory and may lead to a memory saturation problem. We compared execution time and memory consumption in Figure 12. Memory consumption tends to follow a logarithmic trend line because we partially fixed the memory consumption problem by optimising the object representation, but it is not future-proof. This is an implementation limitation, which needs to be addressed in a next version. At the moment, it has been considered acceptable by Michelin.

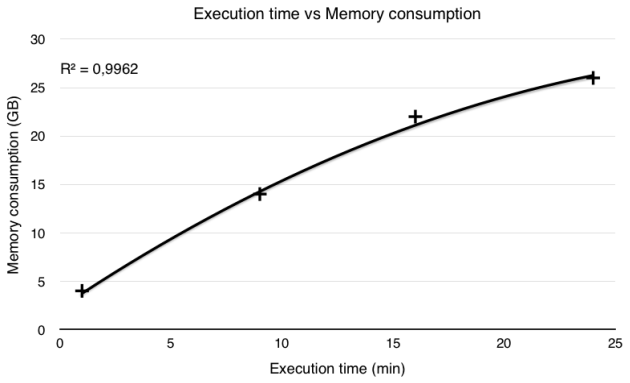

Figure 12: Memory consumption vs execution time

8 Conclusion

This paper has proposed a novel approach to build models of a first production system in order to check whether a second system is conforming to the models. This approach is the result of a collaboration with the manufacturer Michelin to test the CIM2 level of production systems (i.e., the applications controlling devices), without having up-to-date models. As a result, we have presented Autofunk, a fast and scalable framework combining model inference, expert systems and passive testing. From a set of production events, Autofunk generates formal models, which are then reused to test another production system. Conformance is defined with two implementation relations, which take into account that the models may not express all the possible correct behaviours. Initial results on the passive testing method are encouraging, and Michelin engineers see a real potential in this framework.

This framework has several limitations, which deserve further investigations in the future. Our approach can be applied to several kinds of production systems but requires a manual study to fill the prerequisites given in Section 2 and to establish: how to parse production events and the inference rules required to filter the traces. Additionally, if the set of production events is too small, the detection of the entry and exit points cannot be done with K-means. In this situation, these have to be given by an expert of the system. We believe that the use of further data mining approaches on the production events could help in the deduction of some of these requirements.

Our context-specific STS reduction does not appear to be generalisable while keeping the same performance. Another more general solution, which limits over-approximation, is to guide the model inference with the computation of quality metrics [35, 27]. At the moment, these approaches are time-consuming though and can only be applied to small systems because the models are incrementally re-generated from scratch to improve the metrics. In Autofunk, another future direction would be to build sub-models of a production system. By now, we consider a whole workshop as a production system to infer models. Focusing on specific locations of a workshop would allow to build smaller models with a generalisable model inference approach.

Our preliminary results show that, in normal usage, it still remains a big set of possibly failure traces to analyse (we mentioned 2% with our experimentations). Even though the possibly failure trace set remains large, Autofunk eases the work of Michelin engineers by highlighting the traces to focus on. We observed that the larger the initial set of traces is, the less under-approximated the inferred model is, and the less possibly failure traces we have after testing. Yet, the design of an automatic diagnosis method of failure traces could be beneficial.

Acknowledgement

This Research was supported in part by the French National Agency of Research and by the industrial partner Michelin.

References:

[1] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1):4–16, Jan. 2002.

[2] C. Andrès, M. G. Merayo, and M. Nuñez. Formal passive testing of timed systems: Theory and tools. *Softw. Test. Verif. Reliab.*, 22(6):365–405, Sept. 2012.

[3] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87 – 106, 1987.

[4] J. Antunes, N. Neves, and P. Verissimo. Reverse engineering of protocols from network traces. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 169–178, Oct 2011.

[5] J. A. Arnedo, A. Cavalli, and M. Núñez. *Fast Testing of Critical Properties through Passive Testing*, pages 295–310. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[6] E. Bayse, A. Cavalli, M. Nunez, and F. Zaidi. A passive testing approach based on invariants:

application to the {WAP}. *Computer Networks*, 48(2):247 – 266, 2005.

[7] A. Biermann and J. Feldman. On the synthesis of finite-state machines from samples of their behavior. *Computers, IEEE Transactions on*, C-21(6):592–597, June 1972.

[8] E. Brinksma and J. Tretmans. Modeling and verification of parallel processes. chapter Testing Transition Systems: An Annotated Bibliography, pages 187–195. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[9] S. R. Buss. *An introduction to proof theory*.

[10] A. Cavalli, C. Gervy, and S. Prokopenko. New approaches for passive testing using an extended finite state machine specification. *Information and Software Technology*, 45(12):837 – 852, 2003. Testing and Validation of Communication Software.

[11] A. Cavalli, S. Maag, and E. M. de Oca. A passive conformance testing approach for a manet routing protocol. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 207–211, New York, NY, USA, 2009. ACM.

[12] N. Chen and C. Viho. *Passive Interoperability Testing for Request-Response Protocols: Method, Tool and Application on CoAP Protocol*, pages 87–102. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[13] K. T. Cheng and A. S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proceedings of the 30th International Design Automation Conference*, DAC '93, pages 86–91, New York, NY, USA, 1993. ACM.

[14] R. de Nicola and M. C. B. Hennessy. *Automata, Languages and Programming: 10th Colloquium Barcelona, Spain, July 18–22, 1983*, chapter Testing equivalences for processes, pages 548–560. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.

[15] P. Dupont. Incremental regular inference. In *Proceedings of the Third ICGI-96*, pages 222–237. Springer, 1996.

[16] W. Durand and S. Salva. Passive testing of production systems based on model inference. In *13. ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015, Austin, TX, USA, September 21-23, 2015*, pages 138–147. IEEE, 2015.

[17] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[18] Y. Falcone, J.-C. Fernandez, and L. Mounier. *Runtime Verification of Safety-Progress Properties*, pages 40–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[19] L. Frantzen, J. Tretmans, and T. Willemse. Test Generation Based on Symbolic Specifications. In J. Grabowski and B. Nielsen, editors, *FATES 2004*, number 3395 in Lecture Notes in Computer Science, pages 1–15. Springer, 2005.

[20] E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302 – 320, 1978.

[21] V. J. Hodge and J. Austin. A Survey of Outlier Detection Methodologies. *Artificial Intelligence Review*, 22:85–126, 2004.

[22] I. Krka, Y. Brun, D. Popescu, J. Garcia, and N. Medvidovic. Using dynamic execution traces and program invariants to enhance behavioral model inference. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 179–182, New York, NY, USA, 2010. ACM.

[23] J. Lang, P. Liberatore, and P. Marquis. Propositional independence - formula-variable independence and forgetting. *Journal of Artificial Intelligence Research*, 18:391–443, 2003.

[24] D. Lee, D. Chen, R. Hao, R. E. Miller, J. Wu, and X. Yin. Network protocol system monitoring: a formal approach with passive testing. *IEEE/ACM Trans. Netw.*, 14:424–437, April 2006.

[25] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, Aug 1996.

[26] M. Leucker and C. Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009.

The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS07).

[27] D. Lo, L. Mariani, and M. Santoro. Learning extended fsa from software: An empirical assessment. *Journal of Systems and Software*, 85(9):2063 – 2076, 2012. Selected papers from the 2011 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA 2011).

[28] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 501–510, New York, NY, USA, 2008. ACM.

[29] L. Mariani and M. Pezze. Dynamic detection of cots component incompatibility. *IEEE Software*, 24(5):76–85, 2007.

[30] P. Mouttappa, S. Maag, and A. Cavalli. Using passive testing based on symbolic execution and slicing techniques. *Comput. Netw.*, 57(15):2992–3008, Oct. 2013.

[31] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 371–382, Washington, DC, USA, 2009. IEEE Computer Society.

[32] M. Salah, T. Denton, S. Mancoridis, and A. Shokouf. Scenariographer: A tool for reverse engineering class usage scenarios from method invocation sequences. In *In ICSM*, pages 155–164. IEEE Computer Society, 2005.

[33] S. Salva and W. Durand. Autofunk, a fast and scalable framework for building formal models from production systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway, June 29 - July 3, 2015*, pages 193–204, 2015.

[34] S. Salva and W. Durand. Combining model generation and passive testing in the same framework to test industrial systems. Technical report, LIMOS, http://sebastien.salva.free.fr/RR-17-03.pdf, 2017. LIMOS Research report RR-17-03.

[35] P. Tonella, C. D. Nguyen, A. Marchetto, K. Lakhotia, and M. Harman. Automated generation of state abstraction functions using data invariant inference. In *8th International Workshop on Automation of Software Test, AST 2013, San Francisco, CA, USA, May 18-19, 2013*, pages 75–81, 2013.

[36] G. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 3(TR-CTI), 1996.

[37] H. Ural and Z. Xu. An efsm-based passive fault detection approach. In *Proceedings of the 19th IFIP TC6/WG6.1 International Conference, and 7th International Conference on Testing of Software and Communicating Systems*, TestCom'07/FATES'07, pages 335–350, Berlin, Heidelberg, 2007. Springer-Verlag.

[38] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin. Reverse engineering state machines by interactive grammar inference. In *In Proceedings of the 14th Working Conference on Reverse Engineering (WCRE'07*. IEEE, 2007.

[39] C. Wernhard. *Literal Projection for First-Order Logic*, pages 389–402. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[40] C. Wu, F. YuShun, and X. Deyun. *Computer Integrated Manufacturing*, pages 484–529. John Wiley & Sons, Inc., 2007.

[41] X. Wu, V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37, 2008.

[42] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal api rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 282–291, New York, NY, USA, 2006. ACM.

[43] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring specifications for resources from natural language api documentation. *Autom. Softw. Eng.*, 18(3-4):227–261, 2011.